Toshiaki Aoki
Kenji Taguchi (Eds.)

LNCS 7635

Formal Met
and Softwa

14th International Confere
on Formal Engineering Me
Kyoto, Japan, November 20

Lecture Notes in Computer Science 7635

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

Toshiaki Aoki Kenji Taguchi (Eds.)

Formal Methods and Software Engineering

14th International Conference
on Formal Engineering Methods, ICFEM 2012
Kyoto, Japan, November 12-16, 2012
Proceedings

Springer

Volume Editors

Toshiaki Aoki
Japan Advanced Institute of Science and Technology (JAIST)
1-1, Asahidai, Nomi, Ishikawa 923-1292, Japan
E-mail: toshiaki@jaist.ac.jp

Kenji Taguchi
National Institute of Advanced Industrial Science and Technology (AIST)
Nakoji 3-11-46, Amagasaki, Hyogo 661-0974, Japan
E-mail: kenji.taguchi@aist.go.jp

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-34280-6 e-ISBN 978-3-642-34281-3
DOI 10.1007/978-3-642-34281-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012949358

CR Subject Classification (1998): D.2.4, D.2, D.3, F.3, F.4.1, C.2, C.2.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

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

Printed on acid-free paper

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

Preface

The aim of the International Conference on Formal Engineering Methods (ICFEM) is to provide an international forum to discuss research issues in formal methods as well as to bring practitioners and researchers together to further develop and to apply them to real-world problems. Formal methods are now being used in wide range of industrial sectors, and particularly in safety-critical areas such as railways, automobiles, and avionics. We hope that ICFEM plays an important role in encouraging researchers not only to advance theories but also to develop application techniques of formal methods.

This was the 14th conference in the series and the third time that it was held in Japan. It has been over a year since the Great East Japan earthquake and Fukushima nuclear disaster took place. We really appreciate the encouragement, concern, and humanitarian aid from all over the world and would like people to come and see how we quickly recovered from such a great disaster.

The Program Committee received 85 submissions from 30 countries and each paper was reviewed by at least three expert reviewers. We chose 31 papers as the result of intensive discussions held among the Program Committee members. We really appreciate the excellent reviews and lively discussions of the Program Committee members and external reviewers in the review process. This year we chose three prominent invited speakers, Mario Tokoro from Sony Computer Science Laboratories, Robert E. Shostak from Vocera Communications, Inc., and Darren Cofer from Rockwell Collins, Advanced Technology Center. The abstracts of their talks are included in these proceedings.

ICFEM 2012 was jointly organized by the National Institute of Advanced Industrial Science and Technology (AIST) and Japan Advanced Institute of Science and Technology (JAIST). This conference could not have been organized without the strong support from the staff members of both institutes. We would especially like to thank Yuki Chiba, Takashi Kitamura, Kazuhiro Ogata, Weiqiang Kong (University of Kyushu), Kenro Yadake, Hiromi Hatanaka, and Satomi Takeda for their great help in organizing the conference. We also appreciate the gentle guidance and help from General Chairs Shaoying Liu (Hosei University) and Kokichi Futatsugi and the conference chair, Hitoshi Ohsaki.

November 2012

Toshiaki Aoki
Kenji Taguchi

Organization

Steering Committee

Keijiro Araki	Kyushu University, Japan
Michael Butler	University of Southampton, UK
Jin Song Dong	National University of Singapore, Singapore
Jifeng He	East China Normal University, China
Mike Hinchey	University of Limerick, Ireland
Shaoying Liu (Chair)	Hosei University, Japan
Jeff Offutt	George Mason University, USA
Shengchao Qin	University of Teesside, UK

General Chairs

Kokichi Futatsugi	JAIST, Japan
Shaoying Liu	Hosei University, Japan

Conference Chair

Hitoshi Ohsaki	AIST, Japan

Program Chairs

Kenji Taguchi	AIST, Japan
Toshiaki Aoki	JAIST, Japan

Workshop/Tutorial Chairs

Kazuhiro Ogata	JAIST, Japan
Weiqiang Kong	Kyushu University, Japan

Publicity/Publication Chairs

Yuki Chiba	JAIST, Japan
Takashi Kitamura	AIST, Japan

Student Volunteer Chair

Kenro Yadake	JAIST, Japan

Web Chair

Nao Aoki	JAIST, Japan

Conference Secretaries

Hiromi Hatanaka	AIST, Japan
Satomi Takeda	AIST, Japan

Program Committee

Bernhard K. Aichernig	Graz University of Technology, Austria
Cyrille Valentin Artho	AIST, Japan
Richard Banach	University of Manchester, UK
Nikolaj Bjørner	Microsoft Research Redmond, USA
Jonathan P. Bowen	Museophile Limited and London South Bank University, UK
Michael Butler	University of Southampton, UK
Sagar Chaki	Carnegie Mellon Software Engineering Institute, USA
Rance Cleaveland	University of Maryland, USA
Jim Davies	University of Oxford, UK
Zhenhua Duan	Xidian University, China
Joaquim Gabarro	Universitat Politècnica de Catalunya, Spain
Andy Galloway	University of York, UK
Stefania Gnesi	ISTI-CNR, Italy
Wolfgang Grieskamp	Google, USA
Kim Guldstrand Larsen	Aalborg University, Denmark
Klaus Havelund	NASA JPL, California Institute of Technology, USA
Daniel Jackson	MIT, USA
Thierry Jéron	INRIA Rennes, France
Gerwin Klein	NICTA and University of New South Wales, Australia
Weiqiang Kong	Kyushu University, Japan
Peter Gorm Larsen	Aarhus University, Denmark
Insup Lee	University of Pennsylvania, USA
Michael Leuschel	University of Düsseldorf, Germany
Xuandong Li	Nanjing University, China
Yuan-Fang Li	Monash University, Australia
Zhiming Liu	United Nations University, Macao
Dominique Méry	Université de Lorraine, France

Stephan Merz	INRIA Research Center Nancy Grand-Est, France
Huaikou Miao	Shanghai University, China
Alexandre Mota	Universidade Federal de Pernambuco, Brazil
Shin Nakajima	National Institute of Informatics, Japan
Kazuhiro Ogata	JAIST, Japan
Jose Oliveira	Universidade do Minho, Portugal
Jun Pang	University of Luxembourg, Luxembourg
Shengchao Qin	Teesside University, UK
Zongyan Qiu	Peking University, China
S. Ramesh	General Motors R&D, India
Alexander Romanovsky	Newcastle University, UK
Wuwei Shen	Western Michigan University, USA
Marjan Sirjani	Reykjavik University, Iceland
Graeme Smith	University of Queensland, Australia
Jing Sun	The University of Auckland, New Zealand
Jun Sun	Singapore University of Technology and Design, Singapore
Yih-Kuen Tsay	National Taiwan University, Taiwan
Viktor Vafeiadis	Max Planck Institute for Software Systems, Germany
Hai H. Wang	Aston University, UK
Ji Wang	National Laboratory for Parallel and Distributed Processing, China
Wang Yi	Uppsala University, Sweden
Jian Zhang	Chinese Academy of Sciences, China
Huibiao Zhu	East China Normal University, China

Additional Reviewers

Ait Ameur, Yamine
Alpuente, María
Andrews, Zoe
Andronick, June
Barnett, Granville
Bryans, Jeremy
Bu, Lei
Carmona, Josep
Chen, Liqian
Chen, Xin
Chen, Yu-Fang
Chen, Zhenbang
Coleman, Joey
Cunha, Alcino
Daum, Matthias
Dong, Wei
Du, Dehui
Farias, Adalberto Cajueiro De
Ferrari, Alessio
Ferreira, Joao F.
Flodin, Jonas
Fontana, Peter
Ghassemi, Fatemeh
Goré, Rajeev
Gui, Lin
Hasuo, Ichiro
He, Guanhua
Huang, Yanhong
Höfner, Peter
Iliasov, Alexei

Iyoda, Juliano
Juhl, Line
Khakpour, Narges
Khamespanah, Ehsan
Khosravi, Ramtin
Kitamura, Takashi
Li, Qin
Lluch Lafuente, Alberto
Maamria, Issam
Massink, Mieke
Mazzara, Manuel
Melo De Sousa, Simão
Murray, Toby
Nielsen, Claus Ballegaard
Orejas, Fernando
Passmore, Grant
Petrocchi, Marinella
Plagge, Daniel
Proenca, Jose
Ricker, Laurie
Sabouri, Hamideh
Salehi Fathabadi, Asieh
Sanan, David
Sarshogh, Mohammad Reza
Satpathy, Manoranjan
Schäf, Martin
Sharify, Zeynab
Shi, Ling
Shu, Qin
Silva, Renato Alexandre
Singh, Neeraj
Song, Songzheng
Sousa Pinto, Jorge
Stainer, Amelie
Su, Wen
Thomson, Jimmy
Tounsi, Mohamed
Traonouez, Louis-Marie
Tsai, Ming-Hsien
Venkatasubramanian, Krishna K.
Wang, Bow-Yaw
Wang, Linzhang
Wang, Shaohui
Wang, Zheng
Wu, Peng
Xiao, Hao
Xu, Meng
Zhang, Chenyi
Zhang, Nan
Zhang, Pengcheng
Zhang, Yufeng
Zhao, Jianhua
Zhong, Hao
Zhu, Ping

Sponsors

National Institute of Advanced Industrial Science and Technology (AIST), Japan
Research Center for Software Verification, Japan Advanced Institute of Science and Technology (JAIST), Japan

Table of Contents

Invited Speech

Concurrency

Applications of Formal Methods to New Areas

Quantity and Probability

Formal Verification

Modeling and Development Methodology

Temporal Logics

Abstraction and Refinement

Tools

Testing and Runtime Verification

Toward Practical Application of Formal Methods in Software Lifecycle Processes

Mario Tokoro

Sony Computer Science Laboratories, Inc., Tokyo, Japan
mario.tokoro@csl.sony.co.jp

Recent information systems are getting larger and more complex, and are used for a long period of time, continually being modified to meet the unexpected changes of service objectives, users' requirements, available technologies, standards, and regulations. Such systems usually include externally-developed modules, and are often connected to other systems, which may change occasionally. Thus, today's software lifecycle processes must be able to cope with such changes.

One of the important goals of software lifecycle processes is to keep consistency between the stakeholders' aim at a system and its specification, between the specification and its implementation, and between the implementation and expected result of operations on the system, throughout the system's lifecycle in the above-mentioned situations. This is extremely difficult to achieve not just because synchronized update of the aim, specification, implementation, and the expectation of operation result is almost impossible, but also because the meaning of each word cannot be completely defined, which is known as the indeterminacy problem, and may change as the time progresses. The vocabulary may also change. Formal methods play an important role in software lifecycle processes, however, the issues lie not on the logic part per se, but on the interface that facilitates description of the system and accommodates such changes.

We are challenging toward the aforementioned goal in the DEOS (Dependability Engineering for Open Systems) project (http://www.dependable-os.net/osddeos/index-e.html). The DEOS process treats the initial development, the modification of a system, and system operation as an integrated iterative lifecycle process. It includes an extension to assurance cases called D-Case for stakeholders to achieve consensus on dependability issues, and the DEOS architecture which provides flexible monitoring and control functions. D-Case Editor, D-Case Viewer, and D-Case/Agda together facilitate description of the system and accommodate various changes. The Agreement Description Database (D-ADD) retains all the versions of D-Case descriptions with the reasons of revision and the trace of all changes of the meaning of a word and vocabulary. This formation enables formal methods to be practically used in the DEOS software lifecycle process.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, p. 1, 2012.

Formal Methods in the Aerospace Industry: Follow the Money

Darren Cofer

Rockwell Collins, Advanced Technology Center
7805 Telegraph Rd. #100
Bloomington, MN 55438
ddcofer@rockwellcollins.com

Abstract. Modern aircraft contain millions of lines of complex software, much of it performing functions that are critical to safe flight. This software must be verified to function correctly with the highest levels of assurance, and aircraft manufacturers must demonstrate evidence of correctness through a rigorous certification process. Furthermore, the size and complexity of the on-board software are rising exponentially. Current test-based verification methods are becoming more expensive and account for a large fraction of the software development cost. New approaches to verification are needed to cope effectively with the software being developed for next-generation aircraft.

Formal analysis methods such as model checking permit software design models to be evaluated much more completely than is possible through simulation or test. This permits design defects to be identified and eliminated early in the development process, when they have much lower impact on cost and schedule. Advances in model checking technology, the adoption of model-based software development processes, and new certification guidance are enabling formal methods to be used by the aerospace industry for verification of software.

This talk provides an overview of our work applying formal methods, such as model checking, to the development of software for commercial and military aircraft [1]. Formal methods being used to provide increased assurance of correctness, reduce development cost, and satisfy certification objectives. A number of applications of formal methods at Rockwell Collins will be presented to illustrate these benefits and how they relate to the aerospace industry.

The traditional justification for the use of formal methods has been to provide increased assurance of correctness, especially for systems or components that implement safety-critical functions. Model checking excels in this area, providing comprehensive exploration of system behavior and exposure of design errors.

However, the strongest motivation for adoption of model checking in the industry seems much more likely to be cost reduction [2]. The ability to detect and eliminate defects early in the development process has a clear impact on downstream costs. Errors are much easier and cheaper to correct in the requirements and design phases than during subsequent implementation and integration phases.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 2–3, 2012.

An additional benefit which may become increasingly important is the ability to satisfy certification objectives through the use of formal methods, including model checking. New certification guidance supporting the use of formal methods has been included in the recently published DO-178C [3], the industry standard governing software aspects of aircraft certification. This will also impact the economic motivations surrounding the use of formal methods.

References

1. Miller, S., Whalen, M., Cofer, D.: Software Model Checking Takes Off. Communications of the ACM 53(2), 58–64 (2010)
2. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of Formal Analysis into a Model-Based Software Development Process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)
3. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. RTCA (2012)

Applying Term Rewriting to Speech Recognition of Numbers

Robert E. Shostak

Vocera Communications, Inc., San Jose, CA 95126 USA

Abstract. In typical speech recognition applications, the designer of the application must supply the recognition engine with context- free grammars defining the set of allowable utterances for each recognition.

In the case of the recognition of numeric quantities such as phone and room numbers, such grammars can be fairly tricky to formulate. The natural pronunciations of a number may have many variations and special cases depending on the language and country. Consider, for example, the pronunciations of the four-digit phone extension "2200" in the United Kingdom. One could pronounce this "two-two-zero-zero", of course, but you will also hear "twenty-two hundred", "double-two double-naught", and many other combinations. In North America, on the other hand, you would almost never hear the use of "double", "triple", or "naught". Conventions for pronouncing numbers also depend heavily on the type of quantity being recognized. The year 1987, for example, could be pronounced "nineteen eighty-seven" or in literary contexts, "nineteen hundred and eighty seven" - but never "one-nine eighty-seven" or "one-thousand-nine-hundred and eighty-seven".

The richness and idiosyncrasies of pronunciation possibilities, together with the combinatorics of dealing with multi-digit numbers of varying lengths, thus make the practical task of manually designing sufficiently complete number grammars laborious and error prone. This is especially true for applications that must be localized to countries having differing telephone dial plans and conventions for natural pronunciation. Ideally, one would like to be able to generate such grammars automatically, or at least semi-automatically.

Doing so requires an intuitive specification technique that allows one to easily encode pronunciation rules from which a grammar can be automatically derived. In this talk, we will show how to define certain formal term rewriting systems – which we call *Number Generating Term Rewriting Systems* (NGTRS) – that work extremely well for this purpose. We will show that given an NGTRS that generates pronunciations for digit strings representing numeric quantities, we can mechanically generate an equivalent context-free grammar. We'll give some examples in a number of different natural languages, and explain how classical term rewriting proof techniques can be used to verify the consistency and completeness of the construction.

The method we describe was put to real-world use in Vocera's speech-controlled communication system. This is a commercial product that consists of tiny, Star Trek-like wearable communicator badges operating against an enterprise-class server on a Wi-Fi network. The system is currently used by nearly a half-million nurses and doctors every day in hospitals in several countries.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, p. 4, 2012.

Variable Permissions for Concurrency Verification

Duy-Khanh Le, Wei-Ngan Chin, and Yong-Meng Teo

Department of Computer Science, National University of Singapore
{leduykha,chinwn,teoym}@comp.nus.edu.sg

Abstract. In the multicore era, verification for concurrent programs is increasingly important. Although state-of-the-art verification systems ensure safe concurrent accesses to heap data structures, they tend to ignore program variables. This is problematic since these variables might also be accessed by concurrent threads. One solution is to apply the same permission system, designed for heap memory, to variables. However, variables have different properties than heap memory and could benefit from a simpler reasoning scheme. In this paper, we propose a new permission system to ensure safe accesses to shared variables. Given a shared variable, a thread owns either a full permission or no permission at all. This ensures data-race freedom when accessing variables. Our goal is to soundly manage the transfer of variable permissions among threads. Moreover, we present an algorithm to automatically infer variable permissions from procedure specifications. Though we propose a simpler permission scheme, we show that our scheme is sufficiently expressive to capture programming models such as POSIX threads and Cilk. We also implement this new scheme inside a tool, called VPERM, to automatically verify the correctness of concurrent programs based on given pre/post-specifications.

Keywords: Variable, permission, concurrency, verification.

1 Introduction

Access permissions have recently attracted much attention for reasoning about heap-manipulating concurrent programs [2, 4, 7, 9–12]. Each heap location is associated with a permission and a thread can access a location if and only if it has the access permission for that location. Permissions can be flexibly transferred among callers and callees of the same threads or among different threads. A thread needs a certain fraction of a permission to read a location but it has to own the full permission in order to perform a write. This guarantees data-race freedom in the presence of concurrent accesses to heap locations.

Program variables, as distinct from heap locations, can also be shared among threads and are prone to data races. Therefore, one may adopt a similar scheme, designed for heap locations, to reason about variables. "Variables as resource" [3, 22] indeed uses such a permission scheme for variables. Each variable x is augmented with a predicate $Own(x, \pi)$ where π denotes the permission to access x. The permission domain is either (0,1] for fractional permissions [4] or [0,∞) for

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 5–21, 2012.

counting permissions [2]. This allows variables to be treated in the same way as heap locations. However, this permission scheme is more complex and places higher burden on programmers to figure out the fraction to be associated to a variable and how to perform permission accounting properly [2]. To the best of our knowledge, we are not aware of any existing verifiers that have fully implemented the idea. SMALLFOOT [1] uses side-conditions to outlaw conflicting accesses to variables. This, however, requires subtle, global, and hard-to-check conditions that a compiler should ensure [3, 23]. Similarly, CHALICE [16, 17], a program verifier developed for concurrency verification, does not support permissions for variables in method bodies. Even VERIFAST [12, 13], a state-of-the-art verifier, still does not naturally support concurrency reasoning using variables, though it has support for variables by simulating them as heap locations. Consequently, existing verification systems narrow the programmers' choice to heap locations instead of variables for shared accesses by concurrent threads at the expense of losing the expressivity and simplicity that variables provide.

In this paper, we argue that variables with their own characteristics could be treated in a much simpler way than heap locations. Firstly, each variable is distinct; therefore, aliasing issue required for heap locations can be ignored for variables in most cases. Secondly, if several threads need to concurrently *read* a variable, the main thread holding the full permission of the variable can just give each child thread a copy of the variable through pass-by-value mechanism. If concurrent threads require *write* access to the same variable, this shared variable can be protected by a mutex lock whose invariant holds the full permission of the variable. Lastly, if only one thread requires a write access to a given variable, we can simply pass the full permission of the variable into the thread (through pass-by-reference) whose permission is only returned when the child thread joins the main thread. This scheme allows concurrent but race-free accesses to variables.

Nonetheless, there are two scenarios where the above scheme is inadequate. The first scenario occurs in languages such as C/C++ when some variables can be aliased through the use of the address-of operator &. The second scenario occurs when concurrent threads require phased accesses to shared variables, e.g. concurrent threads safely read prior to writing to shared variables. In both scenarios, we propose to automatically translate the affected variables into pseudo-heap locations where a more complex heap permission scheme is utilized.

Because of the above observations, we propose to simply assign a permission of either full or zero to a variable. We can utilize heap (or pseudo-heap) locations to complement our concurrent programming model, *where necessary*, and also readily use variables, *where sufficient*. The net result is a rich but still verifiable programming paradigm for concurrent threads. We shall show that our treatment of variable permissions is sound and expressive to capture programming models such as POSIX threads [5] and Cilk [8]. To relieve programmers from annotation efforts, we shall demonstrate an algorithm to automatically infer variable permissions by only looking at procedure specifications. We shall also provide a translation scheme to handle the variable aliasing (that can also be used for variables requiring phased accesses) and thus complement our treatment of variable permissions.

Contributions. In this paper, we make the following contributions:

- A simpler treatment of variable permissions to ensure safe concurrent accesses to program variables, as distinct from heap locations (Section 2 and 4.1). We also demonstrate the applicability of our scheme to popular programming models such as POSIX threads and Cilk (Section 5.1).
- An algorithm to automatically infer variable permissions from procedure specifications. This helps to reduce program annotations (Section 4.2).
- A translation scheme to eliminate variable aliasing for the purpose of program verification. (Section 4.3). We present how to translate programs with pointers and address-of operator (&) into our core language (Section 3).
- A prototype system, VPERM[1], to show that our variable permission scheme is practical to be implemented and to automatically verify concurrent programs such as parallel mergesort and parallel quicksort among others. Experimental results show that our system minimizes user annotations that are typically required in verification (Section 6).

2 Motivating Example

This section illustrates our treatment of variable permissions to reason about concurrent programs. Figure 1 shows an example illustrating the widely-used task-decomposition pattern in concurrent programming. The `main` procedure invokes the `creator` procedure to create a concurrent task and later performs a join to collect its result. In this example, the `main` procedure creates two local variables `x` and `y` and passes them to the `creator`. The `creator` forks a child thread that increases `x` by 1, and itself increases `y` by 2. The identifier `tid` of the child thread is returned to the `main` procedure which will later perform a `join`.

This example shows a fairly complicated inter-procedural passing of variables between the main thread and the child thread. It poses two challenges: (i) how to describe the fact that any accesses to `x` after forking the child thread and before joining it are unsafe, and (ii) how to propagate this fact across procedure boundaries. These issues can be resolved soundly and modularly by our proposed variable permissions.

Modular reasoning is achieved by augmenting the specification of the program with variable permissions: @*full*[...] and @*value*[...]. In pre-conditions (specified after `requires` keyword), @*full*[v^*] and @*value*[v^*] denote lists of pass-by-reference and pass-by-value parameters respectively. If a variable is passed by reference, the caller transfers the full permission of that variable to the callee. If a variable is passed by value, only a copy of that variable is passed to the callee and the caller still has the full permission of that variable. In post-conditions (after `ensures` keyword), @*full*[v^*] specifies the transfer of full permissions from the callee back to the caller via pass-by-reference parameters. Note that callers and callees can be in a single thread in case of normal procedure calls or in different threads in case of asynchronous calls via fork/join.

[1] The tool is available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/vperm/`.

In this example, the `main` procedure transfers the full permissions of `x` and `y` to the `creator` (specified in its precondition as $@full[x, y]$). When forking a new child thread executing the `inc` procedure, the main thread transfers the full permission of `x` to the child thread (using pass-by-reference mechanism). This effect can be seen in the post-condition of the `creator` where we have two concurrent threads separated by the **and** keyword: after giving up the full permission of `x`, the main thread retains the full permission of `y` ($@full[y]$) while the child thread (with identifier **thread**=tid) holds the full permission of `x` ($@full[x]$). Thus, prior to invoking a `join` to merge back the child thread, the main thread has zero permission of `x` and is not allowed to access it (neither read nor write). This ensures data-race freedom since only one thread at a time can have the full permission of `x`.

```
void inc(ref int i, int j)
requires @full[i] ∧ @value[j]
ensures @full[i] ∧ i'=i+j;
{ i = i + j; }

int creator(ref int x, ref int y)
requires @full[x, y]
ensures @full[y] ∧ y'=y+2 ∧ res=tid
  and @full[x] ∧ x'=x+1 ∧ thread=tid;
{
  int tid = fork(inc, x, 1);
  inc(y, 2);
  return tid;
}

void main()
{
  int id;
  int x = 0, y = 0;
  id = creator(x, y);
  ...
  join(id);
  assert (x' + y' = 3);
}
```

Fig. 1. A Motivating Example

In the specification, we use the reserved keyword **thread** to capture the identifier `tid` of a child thread and the keyword **res** to represent the return value of a procedure call (in case of `creator`, the return value is the thread identifier `tid` of the child thread). Additionally, we use *primed notation* to handle updates to variables. The primed version x' of a variable x denotes its latest value; the unprimed version x denotes its initial value (i.e. its value at the beginning of the procedure). Note that a variable x and its primed version x' can be related but are two different logical variables.

One may think that this treatment of variable permissions can be easily captured through parameter passing, e.g. for each reference parameter `v`, just add an $@full[v]$ in the main thread of both pre- and post-conditions. However, this simple assumption may not hold in the context of concurrency. The key question is which thread holds full permission of a given variable. The full permission can belong to the main thread in the pre-condition but later it is transferred to a child thread in the post-condition and vice versa. For example, in the `creator`, the main thread has $@full[x]$ in the pre-condition but this permission is later transferred to the child thread in the post-condition. In summary, the goal of our scheme is to succinctly manage the transfer of variable permissions among threads in a sound and modular manner.

3 Programming and Specification Languages

Our core programming language (Figure 2) is an imperative language with fork/join concurrency for dynamic thread creation. We chose fork/join as constructs for concurrency because they are often used in concurrent programming [18]. A program consists of a list of data declarations ($data\_decl^*$), a list of global variable declarations ($global\_decl^*$), and a list of procedure declarations ($proc\_decl^*$). Each procedure $proc\_decl$ is annotated with pairs of pre/post specifications (Φ_{pr}/Φ_{po}). A parameter $param$ can be passed by value or by reference (**ref**). A **fork** receives a procedure name pn, a list of parameters v^*, and returns a unique thread identifier as an integer. A **join** requires a thread identifier to join the thread back. The semantics of other program statements is standard as can be found in well-known languages such as C/C++. Note that the core language does not include program pointers and address-of operator (&). In Section 4.3, we show how to translate those constructs into the core language.

P	::= $data\_decl^*$ $global\_decl^*$ $proc\_decl^*$	Program
$data\_decl$	::= **data** C { $field\_decl^*$ }	Data declaration
$field\_decl$	::= $type\ f$;	Field declaration
$global\_decl$	::= **global** $type\ v$	Global variable declaration
$proc\_decl$	::= $ret\_type\ pn(param^*)$ $spec^*$ { e }	Procedure declaration
$spec$	::= **requires** Φ_{pr} **ensures** Φ_{po};	Pre/Post-conditions
$param$	::= $type\ v$ | **ref** $type\ v$	Parameter
$type$	::= **int** | **bool** | C	Type
e	::= v | $v.f$ | k	Variable/field/constant
$stmt$	::= $v = \textbf{fork}(pn, v^*)$ | **join**(v) | $pn(v^*)$ | ...	Statement

Fig. 2. Programming Language with Annotations and Concurrency

Shape predicate	$spred$	::=	$[\texttt{self}::]c[(\texttt{f})]\langle v^*\rangle \equiv \Phi$ [**inv** π_0]
Separation formula	Φ	::=	$\bigvee(\exists v^* \cdot \mu[(\textbf{and}\ \mu)^*])^*$
Thread formula	μ	::=	$\kappa \wedge \nu \wedge \gamma \wedge \phi$
Heap formula	κ	::=	**emp** | ℓ | $\kappa_1 * \kappa_2$
Atomic heap formula	ℓ	::=	$p::c[(\texttt{f})]\langle v^*\rangle$
Vperm formula	ν	::=	$@zero[v^*]$ | $@full[v^*]$ | $@value[v^*]$ | $\nu_1 \wedge \nu_2$ | $\nu_1 \vee \nu_2$
Thread id formula	γ	::=	**thread** $= v$ | $true$
Pure formula	ϕ	::=	...
Fractional permission variable $\texttt{f} \in (0,1]$			$v \in$ Variables
$c \in$ Data or predicate names			$k \in$ Integer constants

Fig. 3. Grammar for Specification Language

Figure 3 shows our rich specification language for concurrent programs manipulating variables and heap locations. For variables, we use variable permissions. For heap locations, we support user-defined predicates *spred* [19] and fractional

permissions f [4]. Φ is a separation logic formula [24] in disjunctive normal form. Each disjunct in Φ consists of a thread formula μ for a main thread and a list of thread formulas (separated by the **and** keyword) to represent concurrent threads. Each thread formula μ contains four parts: a heap formula κ, a vperm formula ν, a threading formula γ, and a pure formula ϕ. A *heap formula* κ consists of multiple atomic heap formulas ℓ connected with each other via separation connectives $*$. An atomic heap formula $p{::}c[(\mathtt{f})]\langle v^* \rangle$ represents the fact that a thread has certain permission `f` to access a heap location of type `c` pointed to by `p`. Vperm formula ν describes permissions of variables (Section 4.1). A thread id formula γ specifies the identifier of a concurrent thread using the keyword **thread**; a main thread has a thread id formula of `true`. A pure formula ϕ consists of standard equality/inequality, Presburger arithmetic and set constraints.

4 Variable Permissions for Safe Concurrency

4.1 Verification Rules

Our verification system is based on entailment checking:

$$\Delta_A \vdash \Delta_C \rightsquigarrow \Delta_R$$

Intuitively, the entailment checks if the antecedent Δ_A is precise enough to imply the consequent Δ_C, and computes the residue for the next program state Δ_R.

Formalism. In order to ensure safe concurrent accesses to variables, we use two key annotations for variable permissions:

- $@full[v^*]$ specifies the full permissions of a list of variables v^*. In pre-conditions, it means that v^* is a list of pass-by-reference parameters. In post-conditions, it captures the return of permissions to caller.
- $@value[v^*]$ only appears in pre-conditions to specify a list of pass-by-value parameters v^*.

$$@full[S] \ \wedge \ v{\not\in}S \vdash @full[v] \ \rightsquigarrow \ fail \qquad \textbf{FAIL-1}$$

$$@full[S] \ \wedge \ v{\not\in}S \vdash @value[v] \ \rightsquigarrow \ fail \qquad \textbf{FAIL-2}$$

$$\frac{v \in S}{@full[S] \vdash @full[v] \ \rightsquigarrow \ @full[S{-}\{v\}]} \qquad \textbf{P-REF}$$

$$\frac{v \in S}{@full[S] \vdash @value[v] \ \rightsquigarrow \ @full[S]} \qquad \textbf{P-VAL}$$

$$@full[S_1] \wedge @full[S_2] \ \rightsquigarrow \ @full[S_1 \cup S_2] \qquad \textbf{NORM-1}$$

$$@full[S_1] \vee @full[S_2] \ \rightsquigarrow \ @full[S_1 \cap S_2] \qquad \textbf{NORM-2}$$

$$@full[S_1] \wedge @value[S_2] \ \rightsquigarrow \ @full[S_1 \cup S_2] \qquad \textbf{BEGIN}$$

Fig. 4. Entailment Rules on Variable Permissions

Variable permissions can be transferred among callers and callees of the same thread, and among distinct threads. The verification rules for variable permissions are shown in Figure 4. A main thread (or a caller) that does not have full permission of a variable cannot pass that full permission to another thread

(or a callee) either by reference or by value (**FAIL−1** and **FAIL−2**). After passing a variable by reference, a main thread (or a caller) loses the full permission of that variable (**P−REF**). However, for a pass-by-value variable, it will still retain the full permission (**P−VAL**). The normalization rules **NORM−1** and **NORM−2** soundly approximate sets of full permissions. At the beginning of a procedure, a main thread has full permissions of its pass-by-reference and pass-by-value parameters (**BEGIN**). The rules presented are simple, and this is precisely how we would like the readers to feel. Simplicity has its virtue and we hope that this would encourage safer concurrent programs to be written.

In our implementation, we also support $@zero[\cdots]$ as a dual to $@full[\cdots]$ annotation. The former denotes a set of variables that may possibly have zero permission. This is useful for more concise representation since only a small fraction of variables typically lose their permissions temporarily.

Forward Verification. Forward verification is formalized using Hoare's triples of the form $\{\Phi_{pr}\}P\{\Phi_{po}\}$: given a program P beginning in a state satisfying the pre-condition Φ_{pr}, if it terminates, it will do so in a state satisfying the post-condition Φ_{po}. Our forward verification rules are presented in Figure 5. We only focus on three key statements that transfer variable permissions: procedure call, fork and join. Note that the transfer of variable permissions is done via entailments as illustrated in Figure 4. In our system, each program state $\Delta[\Delta_t^*]$ consists of the current state Δ of a main thread and a list of post-states Δ_t^* of child threads. Here post-states refer to states of child threads after they finish execution. These post-states will be merged into the state of the main thread when child threads are joined.

$$\frac{\{P\}\ pn(v^*)\ \{Q\} \quad \Delta \vdash P \rightsquigarrow \Delta_1 \quad \Delta_2 \triangleq \Delta_1 * Q}{\{\Delta[\Delta_t^*]\}\ pn(v^*)\ \{\Delta_2[\Delta_t^*]\}}\ \textbf{CALL}$$

$$\frac{\begin{array}{c}\{P\}\ pn(v^*)\ \{Q\} \qquad \Delta \vdash P \rightsquigarrow \Delta_1 \\ \Delta_{tnew} \triangleq Q \wedge \mathbf{thread}{=}unique\_id \\ \Delta_2 \triangleq \Delta_1 \circ_{\{v\}} v'{=}unique\_id\end{array}}{\{\Delta[\Delta_t^*]\}\ v := \mathbf{fork}(pn,v^*)\ \{\Delta_2[\Delta_{tnew}::\Delta_t^*]\}}\ \textbf{FORK}$$

$$\frac{\begin{array}{c}(\Delta_1 \wedge \mathbf{thread}{=}id) \in \Delta_t^* \\ \Delta \vdash v' = id \rightsquigarrow \Delta_2 \qquad \Delta_3 \triangleq \Delta_2 * \Delta_1 \\ \Delta_{tnew}^* = \Delta_t^* - [\Delta_1 \wedge \mathbf{thread}{=}id]\end{array}}{\{\Delta[\Delta_t^*]\}\ \mathbf{join}(v)\ \{\Delta_3[\Delta_{tnew}^*]\}}\ \textbf{JOIN}$$

Fig. 5. Forward Verification Rules for Concurrency

In order to perform a procedure call (**CALL**), a main thread should be in a state Δ that can entail the pre-condition P of the procedure pn. For clarity of presentation, we omit the substitutions that link actual and formal parameters of the procedure prior to the entailment. After the entailment, the main thread subsumes the post-condition Q of the procedure with the residue Δ_1 to form a new state Δ_2. The list of concurrent threads Δ_t^* remains unchanged.

Similarly, in order to fork a new child thread (**FORK**), a main thread should be in a state Δ that can satisfy the pre-condition P of the forked procedure pn. Then a new thread Δ_{tnew} with a unique identifier carrying the post-condition Q of the corresponding forked procedure is created. The new thread is then added to the list of child threads. The main thread keeps the identifier of the child thread in its new state Δ_2 via the return value v of the **fork** call.

In the opposite way, when joining a child thread with an identifier v (**JOIN**), the main thread checks if v is a certain identifier in any child thread, merges the post-state of the child thread Δ_1 into its residue state Δ_2 to form a new state Δ_3, and removes the thread from the list of concurrent threads (denoted by the subtraction "$-$"). The rest of verification rules used in our system only operate on the state of the main thread and are standard as discussed in [19].

Theorem 1 (Soundness of Variable Permission Scheme). *Given a program with a set of procedures P^i and their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}) enhanced with variable permissions, if our verification system derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}\ P^i\ \{\Phi^i_{po}\}$ is valid, then the program is free from data races.*

Proof. By proving that the scheme maintains the invariant that the full permission of each variable belongs to at most one thread at any time. More details are given in the technical report [15]. □

4.2 Inferring Variable Permissions

In this section, we investigate inference for variable permissions. Approaches in permission inference for variables [23] and heap locations [7, 10] require entire program code and/or its specifications for their global analysis. The simplicity of our variable permission scheme offers opportunities for automatically and modularly inferring variable permissions by only looking at procedure specifications.

Our inference is based on following key observations. Firstly, local variables of a procedure cannot escape from their lexical scope; therefore, they are not allowed to appear in post-conditions. Secondly, scopes of pass-by-value parameters are only within their procedures; therefore, $@value[...]$ only exists in pre-conditions and updates to these parameters need not be specified in post-conditions. Thirdly, for each procedure with its *R-complete* pre/post-conditions, updates to its reference parameters must be specified in its post-condition via *primed notations*. Lastly, because child threads carry the post-conditions of their corresponding forked procedures, their states include information about updates to variables that were passed by reference to their forked procedures.

Definition 1 (Primed Notations and R-complete Specifications). *Primed notations represent the latest values of program variables; unprimed notations denote either logical variables or initial values of program variables. A procedure specification is R-complete if all updates to its pass-by-reference parameters are specified in the pre/post conditions using primed notations.*

Algorithm 1. Inferring variable permissions from procedure specifications

Input: Φ_{pr}, Φ_{po}: pre/post-conditions of a procedure without variable permissions
Input: V_{ref}, V_{val} : sets of pass-by-reference and pass-by-value parameters
Output: Pre/post-conditions with inferred variable permissions

1: $V_{post} := V_{ref}$
2: /*Infer @*full*[...] annotations for post-condition*/
3: **for** each thread Δ in Φ_{po} **do**
4: /*Set of free variables that are updated in Δ using primed notations*/
5: $V_m := \{v : v \in FreeVars(\Delta) \wedge isPrimed(v)\}$
6: **if** $(V_m - V_{post}) \neq \phi$ **then** Error
7: **else**
8: $\Delta := \Delta \wedge @full[V_m]$
9: $V_{post} := V_{post} - V_m$
10: **end if**
11: **end for**
12: /*excluding reference parameters not updated in post-condition*/
13: $V_{pre} := V_{ref} - V_{post}$
14: /*Infer @*full*[...] annotations for pre-condition's child threads*/
15: /*in the same way as with those in post-condition but replace V_{post} by V_{pre}*/
16: **for** each child thread Δ_t in Φ_{pr} **do**
17: ...
18: **end for**
19: For the main thread Δ in Φ_{pr}: $\Delta := \Delta \wedge @full[V_{pre}] \wedge @value[V_{val}]$
20: **return** Φ_{pr},Φ_{po}

We present our inference in Algorithm 1. For each procedure, the algorithm starts inference for the post-condition first. For each thread in the post-condition (either main thread or child thread), the full permissions are inferred by computing those pass-by-reference parameters that are updated in each thread's specification via primed notations. The `if` statement in line 6 detects an error if there are some primed variables that (1) are not reference parameters or (2) belonged to other threads in the previous iterations. The subtraction in line 9 removes from the set of reference parameters V_{post} those variables whose inferred full permissions already belonged to the current thread. This ensures that only one thread in the specification holds the full permission of a variable. Because child threads in the pre-condition carry the post-conditions of their corresponding forked procedures, we infer variable permissions for these child threads in the same way as with those in the post-condition. Note that the main thread is the currently active execution thread; therefore, its state in the pre-condition does not include primed variables. The main thread of the pre-condition holds full permissions of variables whose are updated (specified in the post-condition) and do not belong to any child threads. The subtraction in line 13 is necessary because there are certain variables that are passed by reference but their full permissions do not belong to any threads (see Section 5.1 for more discussions). Finally, permission annotation @*value*[...] of pass-by-value parameters is added into the main thread of the pre-condition. For illustration, we present a running example in Table 1.

Table 1. Inferring variable permissions for procedure `creator` in Figure 1

Input		Intermediate values	Inferred
V_{ref}:={x, y}, V_{val}:={}			
Φ_{po}:=	$y'=y+2 \wedge$ **res**=tid	V_{post}:={y}, V_m:={y}	@$full[y]$
	and $x'=x+1 \wedge$ **thread**=tid;	V_{post}:={x, y}, V_m:={x}	@$full[x]$
Φ_{pr}:=	`true`	V_{pre}:={x, y}	@$full[x, y]$

Theorem 2 (Soundness of Inference Algorithm). *Given a procedure* `P` *with its R-complete pre/post-conditions (Φ_{pr}/Φ_{po}) without variable permissions, and our inference algorithm results in new pre/post-conditions (Φ'_{pr}/Φ'_{po}) with inferred variable permissions, if our verification system derives a proof, i.e.* $\{\Phi'_{pr}\}$ `P` $\{\Phi'_{po}\}$ *is valid, then the procedure* `P` *is free from data races.*

Proof. We first prove that the inferred full permission of each variable belongs to at most one thread in a procedure's *R-complete* specification. Then we prove that with the inferred variable permissions, the procedure is free from data races. Details are given in the technical report [15]. □

4.3 Eliminating Variable Aliasing

In this section, we investigate the problem of variable aliasing. Aliasing occurs when a data location can be accessed through different symbolic names (i.e. variable names). For example in C/C++, variables can be aliased by the use of address-of operator (&). This poses challenges to program verification in general and concurrency verification in particular. Figure 6a shows a problematic example where `p` and `x` are aliased due to the assignment p=&x. After passing `x` by reference to a child thread, although the main thread does not have permission to access `x`, it can still access the value of `x` via its alias `*p` and therefore incurs possible data races. Our goal is to ensure safe concurrent accesses to variables even in the presence of aliasing, e.g. to outlaw racy accesses to the value of `x`.

```
void inc(ref int i, int j)
requires @full[i] ∧ @value[j]
ensures @full[i] ∧ i'=i+j;
{ i = i + j; }

void main()
{
  int x = 0;
  int * p = &x;
  int id = fork(inc, x, 1);
  ...//accesses to *p are racy
  join(id);

}
```

(a) Original Program

```
void inc(int_ptr i, int j)
requires i::int_ptr⟨old_i⟩ ∧ @value[i, j]
ensures i::int_ptr⟨new_i⟩ ∧ new_i=old_i + j;
{ i.val = i.val + j; }

void main()
{
  int_ptr x = new int_ptr(0);
  int_ptr p = x;
  int id = fork(inc, x, 1);
  ...//accesses to p.val or x.val are illegal
  join(id);
  delete(x);
}
```

(b) Translated Program

Fig. 6. An Example of Eliminating Variable Aliasing

We propose a translation scheme to eliminating variable aliasing by unifying pointers to program variables and pointers to heap locations. The translation is automatic and transparent to programmers. We refer to each variable (or parameter) whose &x appears in the program as an *addressable* variable. Intuitively, for each *addressable* variable, our translation scheme transforms it into a pointer to a pseudo-heap location by the following substitution ρ=[int $\mapsto$ int_ptr, &x $\mapsto$ x, x $\mapsto$ x.val]. Our approach covers values of any type (including primitive and data types). For each type t, there is a corresponding type t_ptr to represent the type of pointers to pseudo-heap locations holding a value of type t. The value located at a pseudo-heap location is accessed via its val field (e.g. x.val).

Definition 2 (Pseudo-heap Locations). *Pseudo-heap locations are heap-allocated locations used for verification purpose only. Each pseudo-heap location represents a transformed program variable and captures the original value of the variable in its* val *field.*

Our scheme also translates program pointers into pointers to heap-allocated locations by the following substitution ρ=[int* $\mapsto$ int_ptr, *p $\mapsto$ p.val]. For pointers that point to another pointer, our translation is also applicable, e.g. int** is translated into int_ptr_ptr. The translation scheme ensures that the semantics of the translated program is equivalent to that of the original program. By transforming addressable variables into pseudo-heap locations, reasoning about aliased variables has been translated to reasoning about aliased heap locations which is easier to handle (i.e. using separation logic [24]). For detailed formal discussions, we refer interested readers to our technical report [15].

An example translation is shown in Figure 6b. The addressable variable x of type int is transformed into a pointer to a pseudo-heap location of type int_ptr. The program pointer p becomes a pointer to the location which x refers to. Variable x will then be passed to a child thread. The procedure inc is also translated to reflect the fact that its reference parameter i has been transformed. In the specification, i::int_ptr⟨old_i⟩ represents the fact that i is a variable of type int_ptr pointing to a pseudo-heap location containing certain value old_i. The original value of x is indeed captured in the value of the pseudo-heap location. In the translated program, when the main thread passes variable x to the child thread, the pseudo-heap location that x points to is also passed to the child thread. Therefore, before the child thread joins, the main thread cannot access the pseudo-heap location (e.g. via p.val) because it no longer owns that location. Note that the pseudo-heap location is deleted at the end to prevent memory leak.

We propose this translation for verification purpose *only* and do not recommend it for compilation use due to performance deficiency since accessing heap-allocated locations are typically more costly than program variables. Variable aliasing may also occur via parameter-passing when two reference parameters of a procedure refer to the same actual variable. Our variable permission scheme (as presented in Section 4.1) disallows the possibility because a caller cannot have two full permissions of a variable to pass it by reference twice.

5 Discussion

5.1 Applicability of the Proposed Variable Permissions

In this section, we discuss the application of our variable permission scheme to popular concurrent programming models such as POSIX threads and Cilk.

Pthreads is considered one of the most popular concurrent programming models for C/C++ [5]. In Pthreads, when creating a new child thread, a main thread passes a pointer to a heap location to the child thread. We model this argument passing by giving a copy of that pointer to the child thread. Furthermore, Pthreads uses global variables to facilitate sharing among threads. If several threads need to concurrently read a shared global variable, the main thread holding the full permission of that variable can just give each child thread a copy of that variable through pass-by-value mechanism. If concurrent threads require write access to the same variables, these variables can be protected by mutex locks whose invariants hold full permissions of the variables. This allows concurrent but race-free accesses to shared global variables. In our system, mutable global variables are automatically converted into pseudo reference parameters for each procedure (that uses them) prior to verification. For shared global variables that are protected by mutex locks, although they are converted into pseudo reference parameters, neither of concurrent threads has the variables' full permissions. It is the locks' invariants that capture the full permissions. Permission annotations for these variables are automatically inferred as shown in Section 4.2. Note that Pthreads' mutex locks are heap-allocated and therefore require reasoning over heap locations which is beyond the scope of this paper. We refer interested readers to [9, 11, 12] for detailed discussions.

Cilk is a well-known concurrent programming model originally developed at MIT and recently adopted by Intel [8]. In Cilk, the `spawn` keyword is used to create a new thread and to return the value of the procedure call instead of a thread identifier. Before the child thread ends, any accesses to that return value are unsafe. Our `fork` can have the same effect by passing an additional variable by reference to capture the return value. This guarantees data-race freedom because only the child thread has the full permission of that variable. More importantly, compared with Pthreads, Cilk provides more flexible parameter passing when creating a child thread. Multiple variables can be passed to a child thread either by value or by reference. This flexible passing can be naturally handled by our pass-by-value and pass-by-reference scheme. To the best of our knowledge, our scheme is the first verification methodology for expressing such a flexible parameter passing style.

5.2 Phased Accesses to Shared Variables

Our variable permission is designed as a simpler permission scheme that can be used where sufficient. For immutable variables that are shared by concurrent threads, the general guideline is to pass copies of those variables to the threads to enjoy safe accesses to those copies. Mutable variables can be shared

but should be protected by mutex locks to ensure race-freedom because there are some threads mutating the variables. However, there is still a class of complex sharing patterns that cannot be directly handled by our scheme. For example, a thread holds a certain permission to read a shared variable and is guaranteed that no other threads can modify the variable (read phase). Later, it acquires additional permissions from other threads and/or lock invariants, and combines them into a full permission to modify the shared variable (write phase). This kind of *phased accesses* to shared variables cannot be verified without splitting a full permission into smaller partial permissions. In this case, the thread can hold a partial permission while the rest of permissions belong to other threads and/or lock invariants.

Under this circumstance, we propose to detect those variables that are accessed in a phased way, and transform them into pseudo-heap locations where a more complex reasoning scheme is utilized [9, 11, 12]. The translation is done in a similar way as shown in Section 4.3. As a result, our general guideline is to readily use variables in most cases where the proposed variable permission scheme is sufficient, and to automatically and uniformly transform variables into pseudo-heap locations where necessary, i.e. in complex scenarios such as aliasing and phased accesses.

6 Experimental Results

We have integrated our variable permission scheme, inference algorithm, and translation scheme into a tool called VPERM for verifying concurrent programs (see our technical report [15] for detailed descriptions of these programs). Our variable permission scheme is best compared with approaches in [3, 22, 23] but implementations of these approaches are not available. Therefore, we compare our system with VERIFAST, a state-of-the-art verifier, in terms of annotation overhead ($\frac{LOAnn}{LOC}$) and verification time. Note that VERIFAST does not naturally support permissions for variables but simulates shared variables as heap locations. All experiments were done on a 3.20GHz Intel Core i7-960 processor

Table 2. Annotation Overhead and Verification Time (*Procs* is the number of procedures used in a program; *LOC* stands for "lines of code"; *LOAnn* stands for "lines of annotation"; Times are in seconds)

Program	Procs	LOC	VERIFAST			VPERM		
			LOAnn	Overhead	Time	LOAnn	Overhead	Time
`alt_threading`	3	17	23	135%	0.03	6	35%	0.18
`threads`	8	68	34	50%	0.04	18	26%	0.44
`tree_count`	1	20	We are not aware of corresponding programs in VERIFAST distribution. They could be coded but require much annotation			2	10%	0.31
`tree_search`	1	23				3	13%	1.17
`task_decompose`	3	19				6	32%	0.21
`fibonacci`	2	30				4	13%	0.29
`quicksort`	3	78				10	13%	1.60
`mergesort`	6	104				12	12%	1.48

with 16GB memory running Ubuntu Linux 10.04. Table 2 shows that although slower than VERIFAST, our system is more automatic in the sense that our system requires significantly less annotation overhead. The annotation overhead does not grow with more lines of code because we only require pre/post specifications at the procedure boundary. On average, we require less than three lines of annotation per procedure. This is important to reduce programmers' efforts for annotation. VERIFAST has higher annotation overhead because beside pre/-post specifications, it requires additional annotations (such as which predicate to open/close or which lemma to apply) for each non-trivial command, such as field-access, fork and join. Although we attempted to write annotations for those programs that are not present in VERIFAST distribution, they are by no means trivial. In many cases, writing correct annotations is difficult and time-consuming. Therefore, we believe that our system shows a decent trade-off where it takes longer verification time (machine effort) but requires considerable less manual annotation (human effort).

7 Related Work

In 1970s, Owicki-Gries [21] came up with the very first tractable proof method for concurrent programs that prevents conflicting accesses to variables using side-conditions. However, these conditions are subtle and hard for compilers to check because it involves examining the entire program [3, 23]. Recently, concurrent separation logic (CSL) [20] has been proposed to nicely reason about heap-manipulating concurrent programs but CSL still relies on side-conditions for dealing with variables. SMALLFOOT verifier [1] uses CSL as its underlying logic and therefore suffers from the same limitation. In contrast, our scheme brings variable permissions into the logic and therefore makes it easier to check for conflicting accesses to variables. "Variables as resource" [3, 22] has proposed to apply permission systems [2, 4], originally designed for heap locations, to variables. Recently, Reddy et. al. [23] reformulate the treatment of variables using the system of syntactic control of interference. They share the same idea of applying fractional permissions [4] to variables. However, these more complex permission schemes place higher burden on programmers to figure out the permission fractions used to associate to variables. To the best of our knowledge, we are not aware of any existing verifiers that have fully implemented the idea. CHALICE [16, 17] ignores the treatment of variables in method bodies while VERIFAST [12, 13] simulates variables as heap locations. Although the underlying semantics of HOLFOOT [25] formalizes "variables as resource", its automatic verification system, which is based on SMALLFOOT, does not allow sharing variables using fractional permissions. In contrast, our variable permission scheme is simpler, using either full or zero permissions, but is expressive enough to support popular programming models such as Pthreads [5] and Cilk [8]. Furthermore, while previous approaches assume theoretical programming languages without dynamic thread creation [3, 22, 25] and procedure [23], our variable permission scheme is more practical to be incorporated into VPERM tool and to verify concurrent programs with procedures and dynamic thread creation such as parallel

quicksort and mergesort. We also presented an algorithm to automatically infer variable permissions and therefore reduce programmers' efforts for annotations. There is some work on automatic inference of access permissions in the literature [7, 10] but they only address permissions for heap locations. Reddy et. al. [23] is the very first work on inferring permissions for variables. However, their approach is different from ours. Firstly, while their approach is a two-pass algorithm over entire program syntax tree and proof outline, our approach can infer variable permissions directly from procedure specifications. Secondly, their work targets programs written in a theoretical language without procedures and dynamic thread creation while our approach supports more realistic programs with procedures and fork/join concurrency. Lastly, most work on verification has often disallowed variable aliasing by using side-conditions [20, 21] or via assertions [3, 9]. Therefore, our presented translation scheme to eliminate variable aliasing is orthogonal to their work since we provide a way to transform addressable variables into pointers to pseudo-heap locations, and thus enable reasoning about their behaviors in the same way as heap locations [9, 20]. In contrast to several informal translation tools [6, 14] which attempt to translate C/C++ programs with pointers into Java, we present a translation scheme with its formal semantics. Another difference is that while they focus on language translation, we aim towards facilitating program verification.

8 Conclusion

We have proposed a new permission system to ensure data-race freedom when accessing variables. Our scheme is simple but expressive to capture programming models such as POSIX threads and Cilk. Through a simple permission scheme for variables, we have extended formal reasoning to popular concurrent programming paradigms that rely on variables. We have provided an algorithm to automatically infer variable permissions and thus reduced program annotations. We have also shown a translation scheme to eliminate variable aliasing and to facilitate verification of programs with aliases on variables. Lastly, we have implemented our scheme into a tool, called VPERM, for verifying concurrent programs including parallel quicksort and parallel mergesort.

Acknowledgement. We thank the reviewers of ICFEM 2012 for insightful feedback. This work is supported by MOE Project 2009-T2-1-063.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
2. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.: Permission Accounting in Separation Logic. In: ACM Symposium on Principles of Programming Languages, pp. 259–270. ACM, New York (2005)

3. Bornat, R., Calcagno, C., Yang, H.: Variables as Resource in Separation Logic. Electronic Notes in Theoretical Computer Science 155, 247–276 (2006)
4. Boyland, J.: Checking Interference with Fractional Permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003)
5. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley Longman Publishing Co., Inc., Boston (1997)
6. Demaine, E.D.: C to Java: Converting Pointers into References. Concurrency - Practice and Experience 10(11-13), 851–861 (1998)
7. Ferrara, P., Müller, P.: Automatic Inference of Access Permissions. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 202–218. Springer, Heidelberg (2012)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation, New York, NY, USA, pp. 212–223 (1998)
9. Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., Sagiv, M.: Local Reasoning for Storable Locks and Threads. In: Shao, Z. (ed.) APLAS 2007. LNCS, vol. 4807, pp. 19–37. Springer, Heidelberg (2007)
10. Heule, S., Leino, K.R.M., Müller, P., Summers, A.J.: Fractional Permissions Without the Fractions. In: Proceedings of the International Workshop on Formal Techniques for Java-like Programs (July 2011)
11. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle Semantics for Concurrent Separation Logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
12. Jacobs, B., Piessens, F.: Expressive modular fine-grained concurrency specification. In: ACM Symposium on Principles of Programming Languages, New York, NY, USA, pp. 271–282 (2011)
13. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
14. Laffra, C.: A C++ to Java Translator. In: Advanced Java: Idioms, Pitfalls, Styles and Programming Tips, ch. 4. Prentice Hall Computer Books (1996)
15. Le, D.K., Chin, W.N., Teo, Y.M.: Variable Permissions for Concurrency Verification. Technical report, National University of Singapore (April 2012)
16. Leino, K.R.M., Müller, P.: A Basis for Verifying Multi-threaded Programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
17. Leino, K.R.M., Müller, P., Smans, J.: Verification of Concurrent Programs with Chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007/2008/2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009)
18. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional (2004)
19. Nguyen, H.H., David, C., Qin, S.C., Chin, W.-N.: Automated Verification of Shape and Size Properties Via Separation Logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
20. O'Hearn, P.W.: Resources, Concurrency and Local Reasoning. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 49–67. Springer, Heidelberg (2004)

21. Owicki, S., Gries, D.: Verifying Properties of Parallel Programs: an Axiomatic Approach. Communications of the ACM, 279–285 (1976)
22. Parkinson, M., Bornat, R., Calcagno, C.: Variables as Resource in Hoare Logics. In: IEEE Logic in Computer Science, Washington, DC, USA, pp. 137–146 (2006)
23. Reddy, U.S., Reynolds, J.C.: Syntactic Control of Interference for Separation Logic. In: ACM Symposium on Principles of Programming Languages, New York, NY, USA, pp. 323–336 (2012)
24. Reynolds, J.: Separation Logic: A Logic for Shared Mutable Data Structures. In: IEEE Logic in Computer Science, Copenhagen, Denmark (July 2002)
25. Tuerk, T.: A Separation Logic Framework for HOL. PhD thesis. University of Cambridge (2011)

A Concurrent Temporal Programming Model with Atomic Blocks*

Xiaoxiao Yang[1], Yu Zhang[1], Ming Fu[2], and Xinyu Feng[2]

[1] State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] University of Science and Technology of China, Hefei, China

Abstract. Atomic blocks, a high-level language construct that allows programmers to explicitly specify the atomicity of operations without worrying about the implementations, are a promising approach that simplifies concurrent programming. On the other hand, temporal logic is a successful model in logic programming and concurrency verification, but none of existing temporal programming models supports concurrent programming with atomic blocks yet.

In this paper, we propose a temporal programming model (αPTL) which extends the projection temporal logic (PTL) to support concurrent programming with atomic blocks. The novel construct that formulates atomic execution of code blocks, which we call *atomic interval formulas*, is always interpreted over two consecutive states, with the internal states of the block being abstracted away. We show that the *framing* mechanism in interval temporal logic also works in the new model, which consequently supports our development of an executive language. The language supports concurrency by introducing a loose interleaving semantics which tracks only the mutual exclusion between atomic blocks. We demonstrate the usage of αPTL by modeling practical concurrent programs.

1 Introduction

Atomic blocks in the forms of **atomic**$\{C\}$ or $\langle C \rangle$ are a high-level language construct that allows programmers to explicitly specify the atomicity of the operation C, without worrying how the atomicity is achieved by the underlying language implementation. They can be used to model architecture-supported atomic instructions, such as compare-and-swap (CAS), or to model high-level transactions implemented by software transactional memory (STM). They are viewed as a promising approach to simplifying concurrent programming in a multi-core era, and have been used in both theoretical study of fine-grained concurrency verification [16] and in modern programming languages [6,17,19] to support transactional programming.

On the other side, temporal logic has proved very useful in specifying and verifying concurrent programs [11] and has seen particular success in the temporal logic programming model, where both algorithms and their properties can be specified in the same language [1]. Indeed, a number of temporal logic programming languages have

* This research was supported by NSFC 61100063, 61161130530, 61073040, 61103023, and China Postdoctoral Science Foundation 201104162, 2012M511420.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 22–37, 2012.

been developed for the purpose of simulation and verification of software and hardware systems, such as Temporal Logic of Actions (TLA) [7], Cactus [12], Tempura [10], MSVL [8], etc. All these make a good foundation for applying temporal logic to implement and verify concurrent algorithms.

However, to our best knowledge, none of these languages supports concurrent programming with atomic blocks. One can certainly implement arbitrary atomic code blocks using traditional concurrent mechanism like locks, but this misses the point of providing the abstract and implementation-independent constructs for atomicity declarations, and the resulting programs could be very complex and increase dramatically the burden of programming and verification.

For instance, modern concurrent programs running on multi-core machines often involve machine-based memory primitives such as CAS. However, to describe such programs in traditional temporal logics, one has to explore the implementation details of CAS, which is roughly presented as the following temporal logic formula ("○" is a basic temporal operator):

$$\texttt{lock}(x) \wedge \bigcirc(\ \texttt{if}\ (x{=}old)\ \texttt{then}\ \bigcirc(x{=}new \wedge ret{=}1)\ \texttt{else}\ \bigcirc(ret{=}0) \wedge \bigcirc\bigcirc\texttt{unlock}(x))$$

To avoid writing such details in programs, a naive solution is to introduce all the low-level memory operations as language primitives, but this is clearly not a systematic way, not to mention that different architecture has different set of memory primitives. With atomic blocks, one can simply define these primitives by wrapping the implementation into atomic blocks:

$$CAS \stackrel{\text{def}}{=} \langle \texttt{if}\ (x = old)\ \texttt{then}\ \bigcirc(x = new \wedge ret = 1)\ \texttt{else}\ \bigcirc(ret = 0)\rangle$$

It is sufficient that the language should support consistent abstraction of atomic blocks and related concurrency.

Similar examples can be found in software memory transactions. For instance, the following program illustrates a common transaction of reading a value from a variable x, doing computation locally (via t) and writing the result back to x:

$$\textbf{\_stm\_atomic}\ \{\ t := x\ \textbf{;}\ compute\_with(t)\ \textbf{;}\ x := t\ \}$$

When reasoning about programs with such transactions, we would like to have a mechanism to express the atomicity of executing transactions, without considering how the atomicity is implemented.

Therefore, we are motivated to define the notation of *atomicity* in the framework of temporal logic and propose a new temporal logic programming language αPTL. Our work is based on the interval-based temporal logics [10,3] and we extend interval temporal logic programming languages with the mechanism of executing code blocks *atomically*, together with a novel parallel operator that tracks only the interleaving of atomic blocks. αPTL could facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way. The framing operators and the minimal model semantics inherited from interval temporal logic programming may enable us to narrow the gap between temporal logic and programming languages in a realistic way.

Our contributions are summarized as follows:

- We extend Projection Temporal Logic (PTL) with the notion of *atomicity*, which supports the formulation of atomic executions of code blocks. The semantics is defined in the interval model and an atomic block is always interpreted over two consecutive states, with internal state transitions abstracted away. Such a formalization respects the essence of atomic execution: the environment must not interfere with the execution of the code block, and the internal execution states of atomic blocks must not be visible from outside. In fact, formulas inside atomic blocks are interpreted over separate intervals in our model, and the connection between the two levels of intervals is precisely defined.
- How values are carried through intervals is a central concern of temporal logic programming. We adopt Duan's framing technique using assignment flags and minimal models [3,4], and show that the framing technique works smoothly with the two levels of intervals, which can carry necessary values into and out of atomic blocks — the abstraction of internal states does not affect the framing in our model. The technique indeed supports our development of an executable language based on αPTL.
- We define a novel notion of parallelism by considering only the interleaving of atomic blocks — parallel composition of two programs (formulas) without any atomic blocks will be translated directly into their conjunctions. For instance, $x = 1 \parallel y = 1$ will be translated into $x = 1 \wedge y = 1$, which reflects the fact that accesses from different threads to different memory locations can indeed occur simultaneously. Such a translation enforces that access to shared memory locations must be done in atomic blocks, otherwise the formulas can result in false, which indicates a flaw in the program.
- αPTL not only allows us to express various low-level memory primitives like CAS, but also makes it possible to model transactions of arbitrary granularity. We illustrate the practical use of αPTLby examples.

2 Temporal Logic

We start with a brief introduction to projection temporal logic (PTL), which was proposed for reasoning about intervals of time for hardware and software systems.

2.1 Projection Temporal Logic

PTL terms and formulas are defined by the following grammar:

$$
\begin{array}{lll}
\text{PTL terms:} & e, e_1, \ldots, e_m & ::= x \mid f(e_1, \ldots, e_m) \mid \bigcirc e \mid \ominus e \\
\text{PTL formulas} & p, q, p_1, p_m & ::= \pi \mid e_1 = e_2 \mid \text{Pred}(e_1, \ldots, e_m) \mid \neg p \mid p \wedge q \mid \bigcirc p \\
 & & \quad \mid (p_1, \ldots, p_m)\ \mathsf{prj}\ q \mid \exists x.p \mid p^+
\end{array}
$$

where x is a variable, f ranges over a predefined set of function symbols, $\bigcirc e$ and $\ominus e$ indicate that term e is evaluated on the next and previous states respectively; π ranges over a predefined set of atomic propositions, and $\text{Pred}(e_1, \ldots, e_m)$ represents a

predefined predicate constructed with $e_1, \ldots, e_m$; operators *next*($\bigcirc$), *projection*(prj) and *chop plus* ($^+$) are temporal operators.

Let $\mathcal{V}$ be the set of variables, $\mathcal{D}$ be the set of values including integers, lists, etc., and $\mathcal{P}rop$ be the set of primitive propositions. A *state* s is a pair of assignments (I_{var}, I_{prop}), where $I_{var} \in \mathcal{V} \to \mathcal{D} \cup \{\mathsf{nil}\}$ (nil denotes undefined values) and $I_{prop} \in \mathcal{P}rop \to \{\mathsf{True}, \mathsf{False}\}$. We often write $s[x]$ for the value $I_{var}(x)$, and $s[\pi]$ for the boolean value $I_{prop}(\pi)$. An *interval* $\sigma = \langle s_0, s_1, \ldots \rangle$ is a sequence of states, of which the length, denoted by $|\sigma|$, is n if $\sigma = \langle s_0, \ldots, s_n \rangle$ and ω if σ is infinite. An empty interval is denoted by ϵ. Interpretation of PTL formulas takes the following form: $(\sigma, i, j) \models p$, where σ is an interval, p is a PTL formula. We call the tuple (σ, i, j) an *interpretation*. Intuitively, $(\sigma, i, j) \models p$ means that the formula p is interpreted over the subinterval of σ starting from state s_i and ending at s_j (j can be ω). Interpretations of PTL terms and formulas are defined in Fig.1.

$$
\begin{array}{lcl}
(\sigma,i,j)[x] & = & s_i[x] \\
(\sigma,i,j)[f(e_1,\ldots,e_m)] & = & \begin{cases} f((\sigma,i,j)[e_1],\ldots,(\sigma,i,j)[e_m]) & \text{if } (\sigma,i,j)[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases} \\
(\sigma,i,j)[\bigcirc e] & = & \begin{cases} (\sigma,i+1,j)[e] & \text{if } i < j \\ nil & \text{otherwise} \end{cases} \\
(\sigma,i,j)[\ominus e] & = & \begin{cases} (\sigma,i-1,j)[e] & \text{if } i > 0 \\ nil & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lll}
(\sigma,i,j) \models \pi & \text{iff} & s_i[\pi] = \mathsf{True} \\
(\sigma,i,j) \models e_1 = e_2 & \text{iff} & (\sigma,i,j)[e_1] = (\sigma,i,j)[e_2] \\
(\sigma,i,j) \models \mathrm{Pred}(e_1,\ldots,e_m) & \text{iff} & \mathrm{Pred}((\sigma,i,j)[e_1],\ldots,(\sigma,i,j)[e_m]) = \mathsf{True} \\
(\sigma,i,j) \models \neg p & \text{iff} & (\sigma,i,j) \not\models p \\
(\sigma,i,j) \models p_1 \wedge p_2 & \text{iff} & (\sigma,i,j) \models p_1 \text{ and } (\sigma,i,j) \models p_2 \\
(\sigma,i,j) \models \bigcirc p & \text{iff} & i < j \text{ and } (\sigma, i{+}1, j) \models p \\
(\sigma,i,j) \models \exists x.p & \text{iff} & \text{there exists } \sigma' \text{ such that } \sigma' \stackrel{x}{=} \sigma \text{ and } (\sigma',i,j) \models p \\
(\sigma,i,j) \models (p_1,\ldots,p_m)\ \mathsf{prj}\ q & \text{iff} & \text{if there are } i = r_0 \leq r_1 \leq \ldots \leq r_m \leq j \text{ such that} \\
 & & (\sigma, r_0, r_1) \models p_1 \text{ and } (\sigma, r_{l-1}, r_l) \models p_l \text{ for all } 1 < l \leq m \text{ and} \\
 & & (\sigma', 0, |\sigma'|) \models q \text{ for } \sigma' \text{ given by :} \\
 & & (1)\ r_m < j \text{ and } \sigma' = \sigma \downarrow (r_0,\ldots,r_m) \cdot \sigma_{(r_m+1..j)} \\
 & & (2)\ r_m = j \text{ and } \sigma' = \sigma \downarrow (r_0,\ldots,r_h) \text{ for some } 0 \leq h \leq m. \\
(\sigma,i,j) \models p^+ & \text{iff} & \text{if there are } i = r_0 \leq r_1 \leq \ldots \leq r_{n-1} \leq r_n = j\ (n \geq 1) \text{ such that} \\
 & & (\sigma, r_0, r_1) \models p \text{ and } (\sigma, r_{l-1}, r_l) \models p\ (1 < l \leq n)
\end{array}
$$

Fig. 1. Semantics for PTL Terms and Formulas

Below is a set of syntactic abbreviations that we shall use frequently:

$$
\varepsilon \stackrel{\text{def}}{=} \neg \bigcirc \mathsf{True} \quad \mathsf{more} \stackrel{\text{def}}{=} \neg\varepsilon \quad p_1 \,;\, p_2 \stackrel{\text{def}}{=} (p_1, p_2)\ \mathsf{prj}\ \varepsilon \quad \Diamond p \stackrel{\text{def}}{=} \mathsf{True} \,;\, p \quad \Box p \stackrel{\text{def}}{=} \neg\Diamond\neg p
$$

$$
\mathsf{len}(n) \stackrel{\text{def}}{=} \begin{cases} \varepsilon & \text{if } n = 0 \\ \bigcirc\mathsf{len}(n-1) & \text{if } n > 1 \end{cases} \qquad \mathsf{skip} \stackrel{\text{def}}{=} \mathsf{len}(1) \qquad x := e \stackrel{\text{def}}{=} \bigcirc x = e \wedge \mathsf{skip}
$$

ε specifies intervals whose current state is the final state; an interval satisfying more requires that the current state not be the final state; The semantics of p_1 ; p_2 says that computation p_2 follows p_1, and the intervals for p_1 and p_2 share a common state. Note that *chop* (;) formula can be defined directly by the projection operator. $\Diamond p$ says that

p holds eventually in the future; $\Box p$ means p holds at every state after (including) the current state; $\mathsf{len}(n)$ means that the distance from the current state to the final state is n; skip specifies intervals with the length 1. $x := e$ means that at the next state $x = e$ holds and the length of the interval over which the assignment takes place is 1.

2.2 Framing Issue

Framing concerns how the value of a variable can be carried from one state to the next. Within ITL community, Duan and Maciej [3] proposed a framing technique through an explicit operator ($\mathsf{frame}(x)$), which enables us to establish a flexible framed environment where framed and non-framed variables can be mixed, with frame operators being used in sequential, conjunctive and parallel manner, and an executable version of framed temporal logic programming language is developed [4,18]. The key characteristic of the frame operator can be stated as: $\mathsf{frame}(x)$ *means that variable x keeps its old value over an interval if no assignment to x has been encountered.*

The framing technique defines a primitive proposition p_x for each variable x: intuitively p_x denotes an assignment of a new value to x — whenever such an assignment occurs, p_x must be true; however, if there is no assignment to x, p_x is unspecified, and in this case, we will use a *minimal model* [3,4] to force it to be false. We also call p_x an *assignment flag*. Formally, $\mathsf{frame}(x)$ is defined as follows:

$$\mathsf{frame}(x) \stackrel{\text{def}}{=} \Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x)), \text{ where } \mathsf{lbf}(x) \stackrel{\text{def}}{=} \neg p_x \rightarrow \exists\, b \;:\; (\ominus x = b \wedge x = b)$$

Intuitively, $\mathsf{lbf}(x)$ (looking back framing) means that, when a variable is framed at a state, its value remains unchanged (same as at the previous state) if no assignment occurs at that state. We say a program is framed if it contains $\mathsf{lbf}(x)$ or $\mathsf{frame}(x)$.

3 Temporal Logic with Atomic Blocks

3.1 The Logic αPTL

αPTLextends projection temporal logic with atomic blocks. It can be defined by the following grammar (αPTLterms are the same as in PTL):

$$\begin{aligned} \alpha\text{PTL Formulas:}\quad p, q, p_1, p_m ::= {} & \langle p \rangle \mid \neg p \mid p \wedge q \mid \bigcirc p \mid (p_1, \ldots, p_m) prj\, q \mid \exists x.p \mid p^+ \\ & \pi \mid e_1 = e_2 \mid \mathrm{Pred}(e_1, \ldots, e_m) \end{aligned}$$

The novel construct $\langle \ldots \rangle$ is used to specify atomic blocks with arbitrary granularity in concurrency and we call $\langle p \rangle$ an *atomic interval formula*. All other constructs are as in PTL except that they can take atomic interval formulas.

The essence of atomic execution is twofold: first, the concrete execution of the code block inside an atomic wrapper can take multiple state transitions; second, nobody out of the atomic block can see the internal states. This leads to an interpretation of atomic interval formulas based on two levels of intervals — at the outer level an atomic interval formula $\langle p \rangle$ always specify a single transition between two consecutive states, which the formula p will be interpreted at another interval (the inner level), which we call an *atomic interval* and must be finite, with only the first and final states being exported

to the outer level. The key point of such a two-level interval based interpretation is the exportation of values which are computed inside atomic blocks. We shall show how framing technique helps at this aspect. A few notations are introduced to support formalizing the semantics of atomic interval formulas.

- Given a formula p, let V_p be the set of free variables of p. we define formula $\mathsf{FRM}(V_p)$ as follows:

$$\mathsf{FRM}(V_p) \stackrel{\text{def}}{=} \begin{cases} \bigwedge_{x \in V_p} \mathsf{frame}(x) & \text{if } V_p \neq \emptyset \\ \mathsf{True} & \text{otherwise} \end{cases}$$

  $\mathsf{FRM}(V_p)$ says that each variable in the set V_p is a framing variable that allows to inherit the old value from previous states. $\mathsf{FRM}(V_p)$ is essentially used to apply the framing technique within atomic blocks, and allows values to be carried throughout an atomic block to its final state, which will be exported.
- Interval concatenation is defined by

$$\sigma \cdot \sigma' = \begin{cases} \sigma & \text{if } |\sigma| = \omega \text{ or } \sigma' = \epsilon \\ \sigma' & \text{if } \sigma = \epsilon \\ \langle s_0, ..., s_i, s_{i+1}, ... \rangle & \text{if } \sigma = \langle s_0, ..., s_i \rangle \text{ and } \sigma' = \langle s_{i+1}, ... \rangle \end{cases}$$

- If $s = (I_{var}, I_{prop})$ is a state, we write $s|_{I'_{prop}}$ for the state (I_{var}, I'_{prop}), which has the same interpretation for normal variables as s but a different interpretation I'_{prop} for propositions.

Definition 1 (Interpretation of atomic interval formulas). *Let (σ, i, j) be an interpretation and p be a formula. Atomic interval formula $\langle p \rangle$ is defined as:*

$(\sigma, i, j) \models \langle p \rangle$ *iff there exists a finite interval σ' and I'_{prop} such that* $\langle s_i \rangle \cdot \sigma' \cdot \langle s_{i+1}|_{I'_{prop}} \rangle \models p \wedge \mathit{FRM}(V_p)$.

The interpretation of atomic interval formulas is the key novelty of the paper, which represents exactly the semantics of atomic executions: the entire formula $\langle p \rangle$ specifies a single state transition (from s_i to s_{i+1}), and the real execution of the block, which represented by p, can be carried over a finite interval (of arbitrary length), with the first and final state connected to s_i and s_{i+1} respectively — the internal states of p (states of σ') are hidden from outside the atomic block. Notice that when interpreting p, we use $\mathsf{FRM}(V_p)$ to carry values through the interval, however this may lead to conflict interpretations to some primitive propositions (basically primitive propositions like p_x for variables that take effect both outside and inside atomic blocks), hence the final state is not exactly s_{i+1} but $s_{i+1}|_{I'_{prop}}$ with a different interpretation of primitive propositions. In Section 3.2, we shall explain the working details of framing in atomic blocks by examples.

In the following, we present some useful αPTL formulas that are frequently used in the rest of the paper.

- $p^* \stackrel{\text{def}}{=} p^+ \vee \varepsilon$ (*chop star*): either chop plus p^+ or ε;
- $p \equiv q \stackrel{\text{def}}{=} \Box(p \leftrightarrow q)$ (*strong equivalence*): p and q have the same truth value in all states of every model;
- $p \supset q \stackrel{\text{def}}{=} \Box(p \rightarrow q)$ (*strong implication*) : $p \rightarrow q$ always holds in all states of every model.

In the temporal programming model, we support two minimum execution unit defined as follows. They are used to construct *normal forms*[4] of program executions.

State formulas. State formulas are defined as follows:.

$$ps ::= x = e \mid x \Leftarrow e \mid ps \wedge ps \mid \mathsf{lbf}(x) \mid \mathsf{True}$$

We consider True as a state formula since every states satisfies True. Note that $\mathsf{lbf}(x)$ is also a state formula when $\ominus x$ is evaluated to a value. The state frame $\mathsf{lbf}(x)$, which denotes a special assignment that keeps the old value of a variable if there are no new assignment to the variable, enables the inheritance of values from the previous state to the current state. Apart from the general assignment $x = e$, in the paper we also allow assignments such as $\mathsf{lbf}(x) \wedge x \Leftarrow e$ and $\mathsf{lbf}(x) \wedge x = e$.

Extended state formulas. Since atomic interval formulas are introduced in αPTL, we extend the notion of state formulas to include atomic interval formulas:

$$Qs ::= ps \mid \langle p \rangle \mid Qs \wedge Qs$$

where p is arbitrary PTL formulas and ps is state formulas.

Theorem 1. *The logic laws in Fig. 2 are valid, where Q, Q_1 and Q_2 are PTL formulas.*

$(L1)$ $\bigcirc p \equiv \odot p \wedge \mathsf{more}$
$(L2)$ $\bigcirc p \supset \mathsf{more}$
$(L3)$ $\bigcirc(p \wedge q) \equiv \bigcirc p \wedge \bigcirc q$
$(L4)$ $\bigcirc(p \vee q) \equiv \bigcirc p \vee \bigcirc q$
$(L5)$ $\bigcirc(\exists x : p) \equiv \exists x : \bigcirc p$
$(L6)$ $\Box p \equiv p \wedge \odot \Box p$
$(L7)$ $\Box p \wedge \varepsilon \equiv p \wedge \varepsilon$
$(L8)$ $\Box p \wedge \mathsf{more} \equiv p \wedge \bigcirc \Box p$
$(L9)$ $(p \vee q)\ ;\ ps \equiv (p\ ;\ ps) \vee (q\ ;\ ps)$
$(L10)$ $(ps \wedge p)\ ;\ q \equiv ps \wedge (p\ ;\ q)$
$(L11)$ $\bigcirc p\ ;\ q \equiv \bigcirc(p\ ;\ q)$
$(L12)$ $\varepsilon\ ;\ q \equiv q$
$(L13)$ $\langle Q_1 \vee Q_2 \rangle \equiv \langle Q_1 \rangle \vee \langle Q_2 \rangle$
$(L14)$ $\langle Q_1 \rangle \equiv \langle Q_2 \rangle$, *if* $Q_1 \equiv Q_2$
$(L15)$ $(p_1 \vee p_2)\ ;\ q \equiv (p_1\ ;\ q) \vee (p_2\ ;\ q)$
$(L16)$ $\exists x : p(x) \equiv \exists y : p(y)$
$(L17)$ $x = e \equiv (p_x \wedge x = e) \vee (\neg p_x \wedge x = e)$
$(L18)$ $\mathsf{lbf}(x) \equiv p_x \vee (\neg p_x \wedge (x = \ominus x))$
$(L19)$ $\mathsf{lbf}(x) \wedge x = e \equiv x \Leftarrow e\ \ (\ominus x \neq e)$
$(L20)$ $\mathsf{lbf}(x) \wedge x \Leftarrow e \equiv x \Leftarrow e$
$(L21)$ $\mathsf{frame}(x) \wedge \mathsf{more} \equiv \bigcirc(\mathsf{frame}(x) \wedge \mathsf{lbf}(x))$
$(L22)$ $\mathsf{frame}(x) \wedge \varepsilon \equiv \varepsilon$
$(L23)$ $(Qs \wedge \bigcirc p_1)\ ;\ p_2 \equiv Qs \wedge \bigcirc(p_1\ ;\ p_2)$
$(L24)$ $Q_1 \wedge \langle Q \rangle \equiv Q_2 \wedge \langle Q \rangle$, if $Q_1 \equiv Q_2$
$(L25)$ $\langle Q \rangle \equiv \langle \mathsf{frame}(x) \wedge Q \rangle$ x is a free variale in Q
$(L26)$ $\bigcirc e_1 + \bigcirc e_2 = \bigcirc(e_1 + e_2)$

Fig. 2. Some αPTL Laws

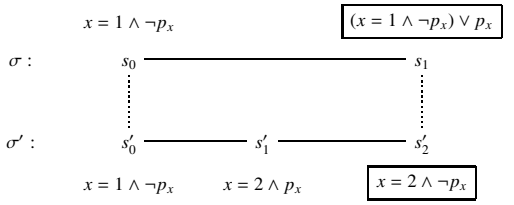
3.2 Support Framing in Atomic Interval Formulas

Framing is a powerful technique that carries values through over interval states in temporal logic programming and it is also used inside atomic blocks in αPTL. While in PTL, framing is usually explicitly specified by users, we have made it inherent in the semantics of atomic interval formulas in αPTL, which is the role of $\mathsf{FRM}(V_Q)$. However, framing inside atomic blocks must be carefully manipulated, as we use same primitive propositions (such as p_x) to track the modification of variables outside and inside

atomic blocks — conflict interpretations of primitive propositions may occur at the exit of atomic blocks. We demonstrate this by the following example:

$$(\sigma, 0, |\sigma|) \models \mathsf{FRM}(\{x\}) \wedge x = 1 \wedge \langle Q \rangle \quad \text{where } Q \stackrel{\text{def}}{=} \bigcirc x = 2 \wedge \mathsf{len}(2)$$

Q requires the length of the atomic interval (that is σ' in the following diagram) to be 2. Inside the atomic block, which is interpreted at the atomic interval, $\mathsf{FRM}(\{x\})$ will record that x is updated with 2 at the second state (s'_1) and remains unchanged till the last state (s'_2). When exiting the atomic block (at state s'_2 of σ'), we need to merge the updating information with that obtained from $\mathsf{FRM}(\{x\})$ outside the block, which intend to inherit the previous value 1 if there is no assignment to x. This conflicts the value sent out from the atomic block, which says that the value of x is 2 with a negative proposition $\neg p_x$. The conflict is well illustrated by the following diagram:



A naive merge of the last states (s_1 and s'_2) in both intervals will produce a false formula: $((x = 1 \wedge \neg p_x) \vee p_x) \wedge (x = 2 \wedge \neg p_x)$. We solve this problem by adopting different interpretation of p_x (the assignment proposition for x) at the exporting state (s_1 in σ and s'_2 in σ'): outside the atomic block (at s_1) we have $(x = 1 \wedge \neg p_x \vee p_x)$ while inside the block we replace s'_2 by $s_1|_{I'_{prop}}$ with a reinterpretation of p_x in I'_{prop}, which gives rise to $(x = 1 \wedge \neg p_x \vee p_x) \wedge (x = 2) \equiv (x = 2) \wedge p_x$ at state s_1. This defines consistent interpretations of formulas outside and inside atomic blocks. It also conforms to the intuition: as long as the value of global variable x is modified inside atomic blocks, then the primitive proposition p_x associated with x must be set True when seen from outside the block, even though there is no assignment outside or at the last internal state, as in the above example.

4 Temporal Programming with Atomic Blocks

This section introduces a temporal programming language based on the logic αPTL, where for concurrent programming, we extend the interval temporal programming with atomic interval formulas, and a *parallel operator*, which captures the interleaving between processes with atomic blocks.

4.1 Expressions and Statements

Expressions. αPTL provides permissible arithmetic expressions and boolean expressions and both are basic terms of αPTL.

Arithmetic expressions: $e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \ \mathsf{op} \ e_1$
Boolean expressions: $b ::= \mathsf{True} \mid \mathsf{False} \mid \neg b \mid b_0 \wedge b_1 \mid e_0 = e_1 \mid e_0 < e_1$

where n is an integer, x is a program variable, op represents common arithmetic operations, $\bigcirc x$ and $\ominus x$ mean that x is evaluated over the next and previous state respectively.

Statements. Figure 3 shows the statements of αPTL, where $p, q, \ldots$ are αPTL formulas. ε means termination on the current state; $x = e$ represents unification over the current

Termination :	ε	Unification :	$x = e$
Positive unification :	$x \Leftarrow e$	Assignment :	$x := e$
State frame :	$\mathsf{lbf}(x)$	Interval frame :	$\mathsf{frame}(x)$
Conjuction statement :	$p \wedge q$	Selection statement:	$p \vee q$
Next statement :	$\bigcirc p$	Sequential statement :	$p \,;\, q$
Conditional statement :	$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q$	Existential quantification :	$\exists x : p(x)$
While statement :	$\texttt{while}\ b\ \texttt{do}\ p$	Atomic block :	$\langle p \rangle$
Parallel statement :	$p \,\|\|\|\, q$	Latency assignment :	$x :=^{+} e$

Fig. 3. Statements in αPTL

state or boolean conditions; $x \Leftarrow e$, $\mathsf{lbf}(x)$ and $\mathsf{frame}(x)$ support framing mechanism and are discussed in Section 3.2; the assignment $x := e$ is as defined in Section 2.1; $p \wedge q$ means that the processes p and q are executed concurrently and they share all the states and variables during the execution; $p \vee q$ represents selection statements; $\bigcirc p$ means that p holds at the next state; $p \,;\, q$ means that p holds at every state from the current one till some state in the future and from that state on p holds. The conditional and while statements are defined as below:

$$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q \stackrel{\text{def}}{=} (b \wedge p) \vee (\neg b \wedge q), \qquad \texttt{while}\ b\ \texttt{do}\ p \stackrel{\text{def}}{=} (p \wedge b)^{*} \wedge \Box(\varepsilon \rightarrow \neg b)$$

We use a renaming method [18] to reduce a program with existential quantification.

The last three statements are new in αPTL. $\langle p \rangle$ executes p atomically. $p \,|||\, q$ executes programs p and q in parallel and we distinguish it from the standard concurrent programs by defining a novel interleaving semantics which tracks only the interleaving between atomic blocks. Intuitively, p and q must be executed at independent processors or computing units, and when neither of them contains atomic blocks, the program can immediately reduce to $p \wedge q$, which indicates that the two programs are executed in a truly concurrent manner. The formal definition of the interleaving semantics will be given in Section 4.3. However, the new interpretation of parallel operator will force programs running at different processors to execute synchronously as if there is a global clock, which is certainly not true in reality. For instance, consider the program

$$(x := x + 1; y := y + 1) \,|||\, (y := y + 2; x := x + 2).$$

In practice, if the two programs run on different processors, there will be data race between them, when we intend to interpret such program as a false formula and indicate a programming fault. But with the new parallel operator $|||$, since there is no atomic blocks, the program is equivalent to

$$(\bigcirc x = x + 1 \wedge \bigcirc(\bigcirc y = y + 1)) \wedge (\bigcirc y = y + 2 \wedge \bigcirc(\bigcirc x = x + 2)),$$

which can still evaluate to true.

The latency assignment $x :=^{+} e$ is introduced to represent the non-deterministic delay of assignments:

$$x :=^{+} e \stackrel{\text{def}}{=} \bigvee_{n \in [1 \ldots N]} \mathsf{len}(n) \wedge \mathsf{fin}(x = e)$$

where $\mathsf{fin}(p) \stackrel{\text{def}}{=} \Box(\varepsilon \rightarrow p)$ and N is a constant denoting a latency bound. The intuition of latency assignment is that an assignment can have an arbitrary latency up to the bound before it takes effect. We explicitly require that *every assignment outside atomic blocks must be a latency assignment*. In order to avoid an excessive number of parentheses, the priority level of the parallel operator is the lowest in αPTL.

4.2 Semi-normal Form

In the αPTL programming model, the execution of programs can be treated as a kind of formula reduction. Target programs are obtained by rewriting the original programs with logic laws (see Fig.2), and the laws ensure that original and target programs are logically equivalent. Usually, our target programs should be represented as αPTL formulas in *normal forms*. In this section, we shall describe how αPTL programs can be transformed into their normal forms. Since we have both state formulas and atomic interval formulas as minimum execution units, we use a different regular form of formulas called *semi-normal form* (SNF for short) to define the interleaving semantics, and it provides an intermediate form for transforming αPTL programs into normal forms.

Definition 2 (Semi-normal Form). An αPTL program is *semi-normal* if it has the following form

$$(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$$

where Qs_{ci} and Qs_{ej} are extended state formulas for all i, j, p_{fi} is an αPTL program, $n_1 + n_2 \geq 1$. P_{fi} is an αPTL program.

If a program terminates at the current state, then it is transformed to $\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon$; otherwise, it is transformed to $\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}$. For convenience, we often call $(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi})$ *future formulas* and $(Qs_{ci} \wedge \bigcirc p_{fi})$ *single future formulas*; whereas $(\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ *terminal formulas* and $(Qs_{ej} \wedge \varepsilon)$ *single terminal formulas*.

4.3 The Interleaving Semantics with Atomic Blocks

In the following we define the parallel statement $(p \,|||\, q)$ based on the SNFs. We first define the interleaving semantics for the single future formula $(Qs \wedge \bigcirc p)$ and the single terminal formula $(Qs \wedge \varepsilon)$ in Definition 3. Note that the interleaving semantics only concerns with atomic blocks, and for non-atomic blocks, the interleaving can be regarded as the conjunction of them. Therefore, the definition is discussed depending on the extended state formula Qs. For short, we use the following abbreviation, where $q_1, \ldots, q_l$ are αPTLformulas:

$$\bigwedge_{i=1}^{l} \langle q_i \rangle \stackrel{\text{def}}{=} \begin{cases} \langle q_1 \rangle \wedge \ldots \wedge \langle q_l \rangle & \text{if } l \geq 1, l \in N \\ \mathsf{True} & \text{if } l = 0 \end{cases}$$

Definition 3. Let ps_1 and ps_2 be state formulas, $Qs_1 \equiv ps_1 \wedge \bigwedge_{i=1}^{l_1} \langle q_i \rangle$ and $Qs_2 \equiv ps_2 \wedge \bigwedge_{i=1}^{l_2} \langle q'_i \rangle$ be extended state formulas $(l_1, l_2 \geq 0)$, T_i denote $\bigcirc p_i$ or ε $(i = 1, 2)$. The interleaving semantics for $(Qs_1 \wedge T_1) \,|||\, (Qs_2 \wedge T_2)$ is inductively defined as follows.

– *case* 1:

$$(Qs_1 \wedge \bigcirc p_1) \;|||\; (Qs_2 \wedge \bigcirc p_2) \stackrel{\text{def}}{=} \begin{cases} (i)\ \ (Qs_1 \wedge Qs_2) \wedge \bigcirc(p_1 \;|||\; p_2), & \text{if } V_{q_i} \cap V_{q'_i} = \varnothing \\ (ii)\ (Qs_1 \wedge ps_2) \wedge \bigcirc(p_1 \;|||\; (\bigwedge_{i=1}^{l_2} \langle q'_i \rangle \wedge \bigcirc p_2)) & \\ \quad \vee & \text{if } V_{q_i} \cap V_{q'_i} \neq \varnothing \\ \quad (Qs_2 \wedge ps_1) \wedge \bigcirc(p_2 \;|||\; (\bigwedge_{i=1}^{l_1} \langle q_i \rangle \wedge \bigcirc p_1)) & \end{cases}$$

– *case* 2:

$$(Qs_1 \wedge \varepsilon) \;|||\; (Qs_2 \wedge T_2) \stackrel{\text{def}}{=} \begin{cases} (i)\ (Qs_1 \wedge Qs_2) \wedge T_2, & \text{if } (l_1 = l_2 = 0) \text{ or } (l_1 = 0 \text{ and } l_2 > 0 \text{ and } T_2 = \bigcirc p_2) \\ (ii)\ Qs_1 \wedge \varepsilon, & \text{if } (l_2 > 0 \text{ and } T_2 = \varepsilon) \\ (iii)\ Qs_2 \wedge T_2, & \text{if } (l_1 \neq 0) \end{cases}$$

– *case* 3: $(Qs_1 \wedge T_1) \;|||\; (Qs_2 \wedge \varepsilon)$ can be defined similarly as *case* 2.

In Definition 3, there are three cases. *Case 1* is about the interleaving between two single future formulas. We have two subcases. On one hand, in *case 1*(*i*): If $V_{q_i} \cap V_{q'_i} = \varnothing$, which means that there are no shared variables in both of the atomic interval formulas, then we can execute them in a parallel way by means of the conjunction construct ($\wedge$). On the other hand, in *case 1*(*ii*): If $V_{q_i} \cap V'_{q_i} \neq \varnothing$, which presents that there are at least one variable that is shared with the atomic interval formulas $\bigwedge_{i=1}^{l_1} \langle q_i \rangle$ and $\bigwedge_{i=1}^{l_2} \langle q'_i \rangle$, then we can select one of them (such as $\bigwedge_{i=1}^{l_1} \langle q_i \rangle$) to execute at the current state, and the other atomic interval formulas (such as $\bigwedge_{i=1}^{l_2} \langle q'_i \rangle$) are reduced at the next state. *Case 2* is about the interleaving between one single terminal formula and one single future formula or between two single terminal formulas. In *case 2*(*i*), if Qs_1 and Qs_2 do not contain any atomic interval formulas (i.e., $l_1 = l_2 = 0$) or there are at least one atomic interval formula in Qs_2 (i.e., $l_2 > 0$) and T_2 is required to be the next statement ($\bigcirc p_2$), then we define $(Qs_1 \wedge \varepsilon) \;|||\; (Qs_2 \wedge T_2)$ as $(Qs_1 \wedge Qs_2 \wedge T_2)$. Since the atomic interval formula $\bigwedge_{i=1}^{n} \langle q_i \rangle$ is interpreted over an interval with at least two states, we have $\bigwedge_{i=1}^{n} \langle q_i \rangle \wedge \varepsilon \equiv \mathsf{False}$. We discuss it in *case 2*(*ii*) and (*iii*) respectively. In *case 2*(*ii*), $l_2 > 0$ and $T_2 = \varepsilon$ means that $Qs_2 \wedge T_2$ is False and we define $(Qs_1 \wedge \varepsilon) \;|||\; (Qs_2 \wedge T_2)$ as $(Qs_1 \wedge \varepsilon)$; In *case 2*(*iii*), $(l_1 \neq 0)$ implies that $Qs_1 \wedge \varepsilon$ is False and $(Qs_1 \wedge \varepsilon) \;|||\; (Qs_2 \wedge T_2)$ is defined as $(Qs_2 \wedge T_2)$. Further, *case 3* can be defined and understood similarly.

In Definition 3, we have discussed the interleaving semantics between the single future formula $(Qs \wedge \bigcirc p)$ and the single terminal formula $(Qs \wedge \varepsilon)$. Further, in Definition 4, we extend Definition 3 to general SNFs.

Definition 4. Let $(\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ and $(\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$ $(n_1 + n_2 \geq 1, m_1 + m_2 \geq 1)$ be general SNFs. We have the following definition.

$$\begin{aligned} & (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon) \;|||\; (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon) \\ \stackrel{\text{def}}{=} & \bigvee_{i=1}^{n_1} \bigvee_{k=1}^{m_1} \left(Qs_{ci} \wedge \bigcirc p_{fi} \;|||\; Qs'_{ck} \wedge \bigcirc p'_{fk}\right) \vee \bigvee_{i=1}^{n_1} \bigvee_{t=1}^{m_2} \left(Qs_{ci} \wedge \bigcirc p_{fi} \;|||\; Qs'_{ek} \wedge \varepsilon\right) \vee \\ & \bigvee_{j=1}^{n_2} \bigvee_{k=1}^{m_1} \left(Qs_{ej} \wedge \varepsilon \;|||\; Qs'_{ck} \wedge \bigcirc p'_{fk}\right) \vee \bigvee_{j=1}^{n_2} \bigvee_{t=1}^{m_2} \left(Qs_{ej} \wedge \varepsilon \;|||\; Qs'_{et} \wedge \varepsilon\right) \end{aligned}$$

Definition 5. Let p and q be αPTL programs. If $p \equiv (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon)$ and $q \equiv (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$, then we have

$$p \,|||\, q \stackrel{\text{def}}{=} (\bigvee_{i=1}^{n_1} Qs_{ci} \wedge \bigcirc p_{fi}) \vee (\bigvee_{j=1}^{n_2} Qs_{ej} \wedge \varepsilon) \,|||\, (\bigvee_{k=1}^{m_1} Qs'_{ck} \wedge \bigcirc p'_{fk}) \vee (\bigvee_{t=1}^{m_2} Qs'_{et} \wedge \varepsilon)$$

Definition 5 means that if programs p and q have SNFs, then the interleaving semantics between p and q is the interleaving between their SNFs.

Theorem 2. For any αPTL program p, there exists a SNF q such that $p \equiv q$.

As discussed before, since introducing atomic interval formulas, semi-normal form is necessary as a bridge for transforming programs into normal forms. In the following, we define normal form for any αPTL program and present some relevant results.

Definition 6 (Normal Form). The normal form of formula p in αPTL is defined as follows:

$$p \equiv (\bigvee_{i=1}^{l} p_{ei} \wedge \varepsilon) \vee (\bigvee_{j=1}^{m} p_{cj} \wedge \bigcirc p_{fj})$$

where $l + m \geq 1$ and each p_{ei} and p_{cj} is state formulas, p_{fj} is an αPTL program.

We can see that normal form is defined based on the state formulas, whereas semi-normal form is defined based on the extended state formulas that includes the atomic interval formulas. By the definition of atomic interval formula in Definition 1, we can unfolding the atomic interval formula into formulas such as $ps \wedge \bigcirc ps'$, where ps and ps' are state formulas, which are executed over two consecutive states. Thus, the extended state formulas can be reduced to the state formulas at last.

Theorem 3. For any αPTL program p, there exists a normal form q such that $p \equiv q$.

Theorem 4. For any satisfiable αPTL program p, there is at least one minimal model.

5 Examples

We give some examples in this section to illustrate how the logic αPTL can be used. Basically, executing a program p is to find an interval to satisfy the program. The execution of a program consists of a series of reductions or rewriting over a sequence of states (i.e., interval). At each state, the program is logically equivalent transformed to normal form, such as $ps \wedge \bigcirc p_f$ if the interval does indeed continue; otherwise it reduced to $ps \wedge \varepsilon$. And, the correctness of reduction at each state is enforced by logic laws. In the following, we give three simple examples to simulate transactions with our atomic interval formulas and present the reductions in details. The first example shows that the framing mechanism can be supported in αPTL, and we can also hide local variables when doing logic programming. In the second example, we use a two-thread concurrent program to demonstrate the reduction with the interleaving operator. Finally, the third example shows αPTL can be used to model fine-grained algorithms as well.

$$\mathsf{frame}(x)\wedge x=0\wedge y=0\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\wedge\mathsf{skip}$$

(L1) $\equiv \mathsf{frame}(x)\wedge x=0\wedge y=0\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\wedge\bigcirc\varepsilon\wedge\mathsf{more}$

(L21) $\equiv x=0\wedge y=0\wedge\bigcirc(\mathsf{frame}(x)\wedge\mathsf{lbf}(x)\wedge\varepsilon)\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle$

(L22) $\equiv x=0\wedge y=0\wedge\bigcirc(\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\varepsilon)\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle$

(L25) $\equiv \ldots\wedge\langle\mathsf{frame}(x)\wedge\mathsf{frame}(y)\wedge\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle$

$\overset{z}{\equiv} \ldots\wedge\langle\mathsf{FRM}(\{x,y,z\})\wedge(z:=x\,;\,x:=z+1\,;\,y:=y+1)\rangle$

$\equiv \ldots\wedge\langle\mathsf{FRM}(\{x,y,z\}\wedge((\bigcirc z=x\wedge\mathsf{skip})\,;\,(\bigcirc x=z+1\wedge\mathsf{skip})\,;\,(\bigcirc y=y+1\wedge\mathsf{skip}))\rangle$

(L11,12) $\equiv \ldots\wedge\langle\mathsf{FRM}(\{x,y,z\}\wedge\bigcirc z=0\wedge\bigcirc(\bigcirc x=z+1\wedge\mathsf{skip}\,;\,(\bigcirc y=y+1\wedge\mathsf{skip}))\rangle$

(L21) $\equiv \ldots\wedge\langle\bigcirc(\mathsf{FRM}(\{x,y,z\}\wedge\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\mathsf{lbf}(z)\wedge z=0\wedge (\bigcirc x=z+1\wedge\mathsf{skip}\,;\,(\bigcirc y=y+1\wedge\mathsf{skip}))\rangle$

(L11,12,17,19) $\equiv \ldots\wedge\langle\bigcirc(\mathsf{FRM}(\{x,y,z\}\wedge(x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\wedge\bigcirc x=z+1\wedge\bigcirc(\bigcirc y=y+1\wedge\mathsf{skip}))\rangle$

(L21) $\equiv \ldots\wedge\langle\bigcirc((x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\wedge\bigcirc(\mathsf{FRM}(\{x,y,z\}\wedge x=1\wedge\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\mathsf{lbf}(z)\wedge\bigcirc y=y+1\wedge\bigcirc\varepsilon))\rangle$

(L18-22) $\equiv \ldots\wedge\langle\bigcirc((x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\wedge\bigcirc((x=1\wedge p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge\neg p_z)\wedge\bigcirc\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\mathsf{lbf}(z)\wedge y=1\wedge\varepsilon))\rangle$

(L18-22) $\equiv \ldots\wedge\langle\bigcirc((x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge p_z)\wedge\bigcirc((x=1\wedge p_x)\wedge(y=0\wedge\neg p_y)\wedge(z=0\wedge\neg p_z)\wedge\bigcirc(x=1\wedge\neg p_x\wedge z=0\wedge\neg p_z\wedge(y=1\wedge p_y)\wedge\varepsilon))))\rangle$

(Def1) $\equiv x=0\wedge y=0\wedge\bigcirc(\mathsf{lbf}(x)\wedge\mathsf{lbf}(y)\wedge\varepsilon)\wedge\bigcirc(x=1\wedge y=1)$

$\equiv (x=0\wedge\neg p_x)\wedge(y=0\wedge\neg p_y)\wedge\bigcirc((x=1\wedge p_x)\wedge(y=1\wedge p_y)\wedge\varepsilon)$

Fig. 4. Rewriting Procedure for Prog_1

Example 1. Suppose that a transaction is aimed to increment shared variable x and y atomically and variable t is a local variable used for the computation inside atomic blocks. The transactional code is shown as below, the atomicity is enforced by the low-level implementations of transactional memory system.

$$\textbf{\_stm\_atomic}\ \{\ t:=x\ ;\ x:=t+1\ ;\ y:=y+1\ \}$$

Let the initial state be $x=0, y=0$. The above transactional code can be defined with αPTL as follows.

$$\mathsf{Prog}_1 \overset{\text{def}}{=} \mathsf{frame}(x)\wedge x=0\wedge y=0\wedge\langle\exists t:\mathsf{frame}(t)\wedge(t:=x;x:=t+1;y:=y+1)\rangle\wedge\mathsf{skip}$$

Fig. 4 presents the detailed rewriting procedure for the above transactional code.

The second example is about money transfer in the bank. Suppose that the money could be transferred from the account A to the account B or from the account B to the account A atomically for n ($n \geq 1$) times. The atomicity is enforced by STM implementations. We use two concurrent threads to implement the task as below.

$$\mathsf{Prog}_2 \stackrel{\text{def}}{=} \mathsf{frame}(i, \mathsf{tmp}_1, \mathsf{tmp}_2, \mathsf{accA}, \mathsf{accB}) \wedge (\mathsf{accA} = m_1) \wedge (\mathsf{accB} = m_2) \wedge (T_1 \,|||\, T_2)$$

$$\begin{array}{l} T_1 \stackrel{\text{def}}{=} (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge \\ \quad \mathtt{while}\ (i \leq n \wedge \mathsf{tmp}_1 \leq m_1)\ \mathtt{do} \\ \quad \{ \\ \quad\ i :=^{+} i + 1\ ; \\ \quad\ \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1)\ ; \\ \quad\ (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle \\ \quad \} \end{array} \qquad \begin{array}{l} T_2 \stackrel{\text{def}}{=} (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \\ \quad \mathtt{while}\ (i \leq n \wedge \mathsf{tmp}_2 \leq m_2)\ \mathtt{do} \\ \quad \{ \\ \quad\ j :=^{+} j + 1\ ; \\ \quad\ \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2); \\ \quad\ (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle \\ \quad \} \end{array}$$

Fig. 5. Money Transfer using Transactions

Example 2. Let T_1 denote the money transfer from the account A to the account B and T_2 denote the money transfer from the account B to the account A. This example can be simulated in αPTL shown in Fig. 5.

In the above example, salary_1, salary_2, m_1, m_2 and n are constants, where salary_1 and salary_2 are money need to transfer; m_1 and m_2 are money in the account, and n denotes the transfer times. The transaction for money transfer in each thread is encoded in the atomic interval formula. To implement Prog_2, by Theorem 2, we first reduce threads T_1 and T_2 into their semi-normal forms, which are presented as follows.

$$T_1 \equiv (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge \bigcirc p_1 \wedge \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1); (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle$$
$$T_2 \equiv (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \bigcirc p_2 \wedge \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2); (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle$$

where p_1 and p_2 are programs that will be executed at the next state and we omit their reduction details here. Further, by Definition 3, we have

$$\begin{array}{rl} T_1 \,|||\, T_2 \stackrel{\text{def}}{=} & (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \\ & \quad \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1); (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle \wedge \\ & \quad\ \bigcirc (p_1 \,|||\, (\bigcirc p_2 \wedge \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2); (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle)) \\ \vee & (i = 0) \wedge (\mathsf{tmp}_1 = \mathsf{salary}_1) \wedge (j = 0) \wedge (\mathsf{tmp}_2 = \mathsf{salary}_2) \wedge \\ & \quad \langle (\mathsf{accB} := \mathsf{accB} - \mathsf{tmp}_2)\ ; (\mathsf{accA} := \mathsf{accA} + \mathsf{tmp}_2) \rangle \wedge \\ & \quad\ \bigcirc (p_2 \,|||\, (p_1 \wedge \langle (\mathsf{accA} := \mathsf{accA} - \mathsf{tmp}_1); (\mathsf{accB} := \mathsf{accB} + \mathsf{tmp}_1) \rangle)) \end{array}$$

Now $T_1 \,|||\, T_2$ is reduced to the semi-normal form like $Q_s \wedge \bigcirc p$. Hence, based on the semi-normal form, we can further reduce Prog_2 to its normal form by interpreting the atomic interval formulas.

In addition to simulating transactional codes, αPTL is able to model fine-grained concurrent algorithms as well. Fine-grained concurrency algorithms are usually implemented with basic machine primitives, whose atomicity is enforced by the hardware. The behaviors of these machine primitives can be modeled with *atomic interval formulas* in αPTL like this: "$\langle \bigvee_{i=1}^{n} (ps_i \wedge \bigcirc ps'_i \wedge \mathsf{skip}) \rangle$". For instance, the machine primitive

CAS(x, old, new; ret) can be defined in αPTL as follows, which has the meaning : if the value of the shared variable x is equivalent to old then to update it with new and return 1, otherwise keep x unchanged and return 0.

$$\begin{aligned}\mathsf{CAS}(x, \mathsf{old}, \mathsf{new}; \mathsf{ret}) &\stackrel{\text{def}}{=} \langle\ \texttt{if}\ (x = \mathsf{old})\ \texttt{then}\ x := \mathsf{new} \wedge \mathsf{ret} := 1)\ \texttt{else}\ \mathsf{ret} := 0\ \rangle \\ &\equiv \langle (x = \mathsf{old}\ \wedge x := \mathsf{new} \wedge \mathsf{ret}\ := 1) \vee (\neg(x = \mathsf{old}\ ) \wedge \mathsf{ret} := 0)\rangle \\ &\equiv \langle (x = \mathsf{old}\ \wedge \bigcirc(x = \mathsf{new} \wedge \mathsf{ret}\ = 1 \wedge \varepsilon)) \vee (\neg(x = \mathsf{old}\ ) \wedge \bigcirc(\mathsf{ret} = 0 \wedge \varepsilon)\rangle\end{aligned}$$

6 Related Works and Conclusions

Several researchers have proposed extending logic programming with temporal logic: Tempura [10] and MSVL [8] based on interval temporal logic, Cactus [12] based on branching-time temporal logic, XYZ/E [14], Templog [1] based on linear-time temporal logic. While most of these temporal languages adopt a Prolog-like syntax and programming style due to their origination in logic programming, interval temporal logic programming languages such like Tempura and MSVL can support imperative programming style. Not only does interval temporal logic allow itself to be executed (imperative constructs like sequential composition and while-loop can be derived straightforwardly in PTL), but more importantly, the imperative execution of interval temporal programs are well founded on the solid theory of framing and minimal model semantics. Indeed, interval temporal logic programming languages have narrowed the gap between the logic and imperative programming languages, and facilitate specifying, verifying and developing reactive systems in a more efficient and uniform way [2,5,13,15].

The fact that none of existing temporal logic programming language supports concurrent logic programming with atomic blocks, motivates our work on αPTL— the new logic allows us to model programs with the fundamental mechanism in modern concurrent programming. We have chosen to base our work on interval temporal logic and introduce a new form of formulas — atomic interval formulas, which is powerful enough to specify synchronization primitives with arbitrary granularity, as well as their interval-based semantics. The choice of PTL also enables us to make use of the *framing* technique and develop an executive language on top of the logic consequently. Indeed, we show that framing works smoothly with our interval semantics for atomicity. In the near future work, we shall focus on the practical use of αPTL and extend the theory and the existing tool MSVL [8] to support the specification and verification of various temporal properties in synchronization algorithms with complex data structure such as the linked list queue [9].

References

1. Abadi, M., Manna, Z.: Temporal logic programming. Journal of Symbolic Computation 8, 277–295 (1989)
2. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying Linearisability with Potential Linearisation Points. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 323–337. Springer, Heidelberg (2011)
3. Duan, Z., Koutny, M.: A framed temporal logic programming language. Journal of Computer Science and Technology 19, 341–351 (2004)

4. Duan, Z., Yang, X., Koutny, M.: Semantics of Framed Temporal Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 356–370. Springer, Heidelberg (2005)
5. Duan, Z., Zhang, N.: A complete axiomatization of propositional projection temporal logic. In: TASE 2008, pp. 271–278. IEEE Computer Science (2008)
6. Harris, T.L., Fraser, K.: Language support for lightweight transactions, pp. 388–402 (2003)
7. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. 16(3), 872–923 (1994)
8. Ma, Y., Duan, Z., Wang, X., Yang, X.: An interpreter for framed tempura and its application. In: Proc. 1st Joint IEEE/IFIP Symp. on Theoretical Aspects of Soft. Eng. (TASE 2007), pp. 251–260 (2007)
9. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. J. Parallel Distrib. Comput. 51(1), 1–26 (1998)
10. Moszkowski, B.C.: Executing temporal logic programs. Cambridge University Press (1986)
11. Pnueli, A.: The Temporal Semantics of Concurrent Programs. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 1–20. Springer, Heidelberg (1979)
12. Rondogiannis, P., Gergatsoulis, M., Panayiotopoulos, T.: Branching-time logic programming: The language cactus and its applications. Computer Language 24(3), 155–178 (1998)
13. Schellhorn, G., Tofan, B., Ernst, G., Reif, W.: Interleaved programs and rely-guarantee reasoning with itl. In: Proc. TIME 2011, pp. 99–106. IEEE Computer Science (2011)
14. Tang, C.: A temporal logic language oriented toward software engineering – introduction to XYZ system (I). Chinese Journal of Advanced Software Research 1, 1–27 (1994)
15. Tian, C., Duan, Z.: Model Checking Propositional Projection Temporal Logic Based on SPIN. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 246–265. Springer, Heidelberg (2007)
16. Vafeiadis, V., Parkinson, M.: A Marriage of Rely/Guarantee and Separation Logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)
17. Yang, N., Adam, W., Adl-Tabatabai, Bach, M., Berkowits, S.: Design and implementation of transactional constructs for C/C++. In: Proceedings OOPSLA 2008, pp. 195–212 (September 2008)
18. Yang, X., Duan, Z.: Operational semantics of framed tempura. Journal of Logic and Algebraic Programming 78(1), 22–51 (2008)
19. Zyulkyarov, F., Harris, T., Unsal, O.S., Cristal, A., Valeroh, M.: Debugging programs that use atomic blocks and transactional memory. In: Proc. PPoPP 2010, pp. 57–66 (2010)

A Composable Mixed Mode Concurrency Control Semantics for Transactional Programs

Granville Barnett[1] and Shengchao Qin[2,3]

[1] School of Engineering and Computing Sciences, Durham University
[2] School of Computing, Teesside University
[3] State Key Lab. for Novel Software Technology, Nanjing University
granville.barnett@durham.ac.uk, s.qin@tees.ac.uk

Abstract. Most software transactional memories employ optimistic concurrency control. A pessimistic semantics, however, is not without its benefits: its programming model is often much simpler to reason about and supports the execution of irreversible operations. We present a programming model that supports both optimistic and pessimistic concurrency control semantics. Our pessimistic transactions, guaranteed transactions (gatomics), afford a stronger semantics than that typically employed by pessimistic transactions by guaranteeing run once execution and safe encapsulation of the privatisation and publication idioms. We describe our mixed mode transactional programming language by giving a small step operational semantics. Using our semantics and their derived schedules of actions (reads and writes) we show that conflicting transactions (atomics) and gatomics are serialisable. We then go on to define schedules of actions in the form of Java's memory model (JMM) and show that the same properties that held under our restrictive memory model also hold under our modified JMM.

1 Introduction

Software transactional memory (STM) [27] has gained considerable traction in recent years and has subsequently been adopted by a number of languages [10, 13]. STM is only an *alternative* to locks. One cannot safely substitute every occurrence of a lock with a transaction and guarantee the original program semantics. This is mainly due to the optimistic concurrency control traditionally employed by STMs. For example, one cannot optimistically execute an irreversible operation and still guarantee consistency. Similarly, optimistic concurrency control is not ideal for executing expensive operations, catering for "hot" regions of memory [29], or for systems with finite resources [4].

One approach of addressing these issues is to permit pessimistic and optimistic transactions to co-exist. Previous literature such as that by McCloskey et al. [21], Ziarek et al. [32], Ni et al. [23], Welc et al. [31] and Sonmez et al. [29] have investigated such approaches, with each providing a different take on how and why pessimism should be introduced into systems already exposing optimistic STM. However, each lack a formal underpinning when addressing two problems:

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 38–53, 2012.

when pessimistic semantics are necessary due to the semantics of the operations being performed, and *how* pessimistic and optimistic modes of concurrency control safely co-exist. The closest work on providing a formal foundation for pessimistic and optimistic transactions was by Koskinen et al. [16] but they treat each in isolation. Other work also exists on the semantics of STM such as that by Abadi et al. [1] but again does not provide a model of co-existence for transactions of differing concurrency control semantics.

The focus of this paper is on presenting an operational semantics for a programming language that supports both optimistic and pessimistic transactions. We partition transactional mediation of accesses to memory into two types: transactions (atomic) which are optimistic, and *guaranteed transactions* (gatomic) which are pessimistic. Atomics under our system have the following properties (in addition to the ACI properties [8]): (i) *word-based*: conflicts are detected at the granularity of memory locations; (ii) *out-of-place*: atomics operate upon a thread-local copy of their dataset which is only observed by other threads should the atomic commit; (iii) *optimistic*: a contention manager [12] determines, at the point when all constituent commands of an atomic have been executed, whether or not the atomic should commit or abort; and (iv) *weakly isolated* [6, 7]: atomic accesses are only isolated w.r.t. other atomic and gatomic accesses. The best comparison of a gatomic w.r.t. the current literature is that a gatomic is an obstinate transaction [23] but is guaranteed to never abort, either prior to, or during its execution, and infers a stronger notion of its dataset. Multiple gatomics can run at any given time provided consistency invariants are maintained, unlike single owner read locks presented by Welc et al. [31]. A constituent operation of a gatomic is guaranteed to only ever run once. Furthermore, gatomics offer a sensible and intuitive encapsulation of the privatisation and publication idioms which are erroneous under some STMs [30]. In addition to being word-based and out-of-place, gatomics entail the following properties: (i) *pessimistic*: contention is resolved at the point of execution; and (ii) *dataset inference*: the transitive closure of reachable objects from those referenced within the gatomic form the dataset of the gatomic. Atomics and gatomics can be freely composed w.r.t. one another.

The algorithm in Fig. 1 uses a gatomic to privatise a list suffix to the invoking thread. Under an atomic semantics the privatisation of the list suffix may not be consistent [30]. Executing `privatiseListSuffix` under a gatomic semantics always maintains memory consistency. For example, given two threads T_1 and T_2, where T_1 and T_2 invoke `list.privatiseListSuffix(5)` and resp. `list.privatiseListSuffix(3)` on a shared list object `list` (a singly linked list) which comprises of the values [1..10], we have either T_1 and T_2 printing [5..10] and resp. [3, 4], or [3..10] and resp. [ ]. Under other STMs [30] this example would require programmer specified logic to explicitly transfer ownership of heap locations to the invoking thread [30], however under a gatomic semantics this process is managed entirely by the underlying system.

Our guiding philosophy can be summarised as follows: optimistic concurrency control (atomic) should be used in *most* cases, but for operations that access

```
class LinkedList {
  // ...
  gatomic privatiseListSuffix(Value v) {
    prev := head;
    curr := prev.next;
    while (curr.value != v) {
      prev := curr; curr := curr.next;
      if (curr == null) goto 9;
    }
    prev.next = null;
    while (curr != null) {
      print(curr.value);
      curr = curr.next;
    } }
  // ...
}
```

Fig. 1. Gatomics guarantee the safety of privatising operations

highly contended memory, execute irreversible operations, demand run once semantics or perform expensive computations, then *on-occasion* pessimistic concurrency control (gatomic) may be preferable [26].

To summarise, we make the following contributions:

- We give a small-step operational semantics (Sect. 2) for an object-oriented programming language that supports atomics and gatomics.
- We define legal schedules of reads and writes issued by atomics and gatomics, first in the form of sequential consistency (SC, Sect. 2.5), and then as part of a modified definition of the Java memory model (JMM, Sect. 3).
- We show that the actions issued by conflicting atomics and gatomics are serialisable both under SC and the JMM. (Sects. 2.6 and 3.)

2 Programming Model

2.1 Programming Language

We present a minimal object-oriented language that supports atomics and gatomics for mediating accesses to memory locations. Atomic and gatomic regions of code can be defined at the granularity of a method or be explicitly scoped.

$$
\begin{array}{lll}
\text{prog} & ::= & \text{cdecl}^*\ (t\ v)^*\ (\text{S}\ ||\ \ldots\ ||\ \text{S}) \\
\text{cdecl} & ::= & \mathsf{class}\ cn\ \{\ (t\ v)^*\ \text{meth}^*\ \} \\
\text{t} & ::= & \text{cn} \mid \text{primitive} \\
\text{meth} & ::= & [\ \mathsf{atomic} \mid \mathsf{gatomic}\ ]\ t\ m((t\ p)^*)\ \{\text{C}\} \\
\text{S} & ::= & (t\ v)^*\ \text{C} \\
\text{C} & ::= & v := x \mid v.f := x \mid v.m(p^*) \mid \mathsf{atomic}\{\text{C}\} \mid \mathsf{gatomic}\{\text{C}\} \mid \text{C;C}
\end{array}
$$

Where v, p and x range over variables, t over types and f over the fields defined by the variable's type. Notable features of our language include the use of atomics ($\mathsf{atomic}\{\mathrm{C}\}$) and gatomics ($\mathsf{gatomic}\{\mathrm{C}\}$) as commands. Classes, methods and method calls are also permitted. Methods can be defined to execute under an atomic, gatomic or non-synchronised semantics. For simplicity of presentation, the above language does not include conditional commands and while loops. Conditional commands cause no extra difficulty in our semantic definitions. While loops can be dealt with via their corresponding tail-recursive methods.

2.2 Program Text Preprocessing

Each invocation of an atomic or gatomic method is wrapped with the synchronisation action (SA, either an atomic or gatomic) defined by the method's signature. A method invocation $v.m(p^*)$ is transformed into $\mathsf{atomic}\{v.m(p^*)\}$ if m is defined as an atomic method. Similarly, the invocation $v.m(p^*)$ is transformed into $\mathsf{gatomic}\{v.m(p^*)\}$ if m is defined to execute under a gatomic semantics.

Nested SAs are flattened by applying the following sequence of phases:

1. *Discovery:* determines the strongest semantics used within the nested SAs. The semantics afforded by a gatomic are stronger than that of an atomic.
2. *Semantic Boosting:* uses the semantics yielded by the discovery phase as the new semantics of the outermost SA.
3. *Flattening:* removes the nested SA semantics from the outermost SA resulting in a single monolithic SA.

To illustrate the use of these phases we now apply them to the command $\mathsf{atomic}\{c_i; \mathsf{gatomic}\{c_j\}\}$:

- *Discovery:* the gatomic is selected as its semantics are stronger than that afforded by an atomic.
- *Semantic Boosting:* the outermost SA (atomic) adopts the semantics identified by the discovery phase resulting in $\mathsf{gatomic}\{c_i; \mathsf{gatomic}\{c_j\}\}$.
- *Flattening:* the nested gatomic is removed resulting in $\mathsf{gatomic}\{c_i; c_j\}$.

The last stage of our preprocessing is to associate a unique identifier, id, with each instance of an atomic and gatomic.

2.3 Preliminaries

State. Our model of state is defined as follows:

$$\mathsf{s} \in \mathsf{Stores} \stackrel{\mathsf{def}}{=} \mathsf{Variables} \to \mathsf{Addresses}$$

$$\mathsf{h} \in \mathsf{Heaps} \stackrel{\mathsf{def}}{=} \mathsf{Addresses} \rightharpoonup_{\mathrm{fin}} \mathsf{ObjVal}$$

$$\mathsf{Variables} \stackrel{\mathsf{def}}{=} \{x, y, ...\} \qquad \mathsf{Addresses} \stackrel{\mathsf{def}}{=} \{0, 1, 2, ...\}$$

$$(\mathsf{s}, \mathsf{h}) \in \mathsf{States} \stackrel{\mathsf{def}}{=} \mathsf{Stores} \times \mathsf{Heaps}$$

Note that s is a function that maps a variable to its address on the heap. A heap h is a partial function that maps an address to an object value $cn[f_1 \mapsto \nu_1, \ldots, f_n \mapsto \nu_n]$ where cn is the type of the object and $\nu_1 \ldots \nu_n$ are the values of the fields $f_1 \ldots f_n$. We use $\mathsf{h}(\mathsf{s}(x)).f$ to access the value of a field f of the object pointed-to by x. We often use σ or δ to denote a state, (s, h).

Configurations. Given a parallel composition $C_1 \| \ldots \| C_n$, the machine configuration P is of the form $\langle T_1 \| \ldots \| T_n, \sigma, \Gamma \rangle$, where $T_1, .., T_n$ are thread configurations, σ is the program state, and Γ records all current instances of SAs.

A thread configuration T is of the form $\langle \tau, C_\tau, \mathsf{s}_\tau, \delta \rangle$, where $\tau \in \mathcal{T} = \{1, .., n\}$ is the thread identifier, C_τ is the command to execute, s_τ is the thread's store mapping, and δ is a copy of the program state. δ entails the thread store and the program state.

Every active SA instance is associated with some metadata of the form $(\mathsf{beg}, \mathsf{cmt}, \mathsf{rs}, \mathsf{ws}, \mathsf{ds}, \mathsf{type})$, where beg and cmt are time stamps representing begin and commit times, rs and ws are read and write sets (sets of addresses), $\mathsf{ds} \stackrel{\mathsf{def}}{=} \mathsf{rs} \cup \mathsf{ws}$ is the dataset, and type is the type of the SA instance, defined as Ψ if the SA is a gatomic or undefined ($\perp$) otherwise. Each component of an SA entry is initially $\perp$. Γ in the program configuration is a mapping that takes a thread identifier and an SA label and returns the metadata associated with the specified SA instance. Metadata facilitates the safe execution of SAs; specifically, the checking of which SAs conflict and which do not. Given two distinct SAs x and y, x and y conflict if x's write set (ws) intersected with y's dataset (ds) yields a non-empty set.

Transition Relations. We model the execution of each thread command using the following transition relation: $T, \sigma, \Gamma \xrightarrow{\lambda} T', \sigma', \Gamma'$, as in Fig. 3. To facilitate the presentation, we also define a set of auxiliary rules of the form $C, \sigma, \gamma_R, \gamma_W \xrightarrow{\lambda_\mathsf{R} \mid \lambda_\mathsf{W}} C', \sigma', \gamma'_R, ' \gamma'_W$ in Fig. 2. Where γ_R and γ_W are incrementally accumulated read and write sets. A transition in the auxiliary rules generates a sequence of reads (λ_R) and/or writes (λ_W). An action that is generated due to a reduction in either the thread or auxiliary rules is termed a *fine-grained action*. λ is used as a metavariable that ranges over fine-grained actions. We model the execution of a parallel composition (Fig. 5) using the following transition relation $P \xrightarrow{\|_{\mathsf{mv} \in I \cup M} \Lambda_\mathsf{mv}} P'$, where P, P' are of the form $\langle T_1 \| \ldots \| T_n, \sigma, \Gamma \rangle$ as defined earlier. The label $\|_{\mathsf{mv} \in I \cup M}\ \Lambda_\mathsf{mv}$ denotes that during the transition from P to P', all actions Λ_mv ($\mathsf{mv} \in I \cup M$) take place concurrently in some arbitrary order. Each action Λ_mv entails a sequence of finer-grained actions, λ, either of the form $\mathsf{tbeg}^\frown \lambda^* {}^\frown \mathsf{tcmt}$ (for atomics) or $\mathsf{gbeg}^\frown \lambda^* {}^\frown \mathsf{gcmt}$ (for gatomics). Where λ^* is a sequence of reads and/or writes.

2.4 Thread Command Semantics

We extend the commands of our language to include the following intermediate constructs to facilitate the presentation of the semantics:

$$\text{C} ::= \ \ldots \mid \mathsf{ablk}(\text{C}, \text{C}) \mid \mathsf{gablk}(\text{C}) \mid \mathsf{ret}(\{v_1, \ldots, v_n\}, \text{C})$$

$$\frac{\mathsf{s}' = \mathsf{s}[v \mapsto \mathsf{s}(x)] \quad \gamma'_R = \gamma_R \cup \{x\} \quad \gamma'_W = \gamma_W \cup \{v\}}{v{:=}x, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda_R \lambda_W} \epsilon, (\mathsf{s}',\mathsf{h}), \gamma'_R, \gamma'_W}\ [\textbf{ASSIGN}]$$

$$\frac{\mathsf{h}' = \mathsf{h}[\mathsf{s}(v) \mapsto (\mathsf{h}(\mathsf{s}(v))[f \mapsto \mathsf{s}(x)])] \quad \gamma'_R = \gamma_R \cup \{x\} \quad \gamma'_W = \gamma_W \cup \{\mathsf{s}(v)\}}{v.f{:=}x, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda_R \lambda_W} \epsilon, (\mathsf{s},\mathsf{h}'), \gamma'_R, \gamma'_W}\ [\textbf{FLD\_UPDATE}]$$

$$\frac{t\ mn(t_1\ p_1, .., t_n\ p_n)\ \{c\} \in \mathsf{methods}(cn) \quad \mathsf{s}_0 = [\mathsf{this} \mapsto \mathsf{s}_\tau(u_0), p_1 \mapsto \mathsf{s}_\tau(u_1), .., p_n \mapsto \mathsf{s}_\tau(u_n)] \quad \mathsf{s}' = \mathsf{push\_frame}(\mathsf{s}_0, \mathsf{s}) \quad \gamma'_R = \gamma_R \cup \{u_1, \ldots, u_n\}}{u_0.mn(u_1..u_n), (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda_{R1} \ldots \lambda_{Rn}} \mathsf{ret}(\{\mathsf{this}, p_1..p_n\}, c), (\mathsf{s}',\mathsf{h}), \gamma'_R, \gamma_W}\ [\textbf{INV1}]$$

$$\frac{c, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c', (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}{\mathsf{ret}(V, c), (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} \mathsf{ret}(V, c'), (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}\ [\textbf{INV2}]$$

$$\frac{\mathsf{s}' = \mathsf{pop\_frame}(V, \mathsf{s})}{\mathsf{ret}(V, \epsilon), (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\epsilon} \epsilon, (\mathsf{s}',\mathsf{h}), \gamma_R, \gamma_W}\ [\textbf{INV3}]$$

$$\frac{c_1, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c'_1, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}{c_1; c_2, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c'_1; c_2, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}\ [\textbf{SEQ1}]$$

$$\frac{c_1, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} \epsilon, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}{c_1; c_2, (\mathsf{s},\mathsf{h}), \gamma_R, \gamma_W \xrightarrow{\lambda} c_2, (\mathsf{s}',\mathsf{h}'), \gamma'_R, \gamma'_W}\ [\textbf{SEQ2}]$$

Fig. 2. Auxiliary Sequential Rules instrumented with Datasets

where ablk(C, C) and gablk(C) are intermediate representations of atomic$\{C\}$ and resp. gatomic$\{C\}$ within the program source text. The second component of ablk is the backup program command and is used as the point to rollback to should the atomic abort. The construct ret$(\{p_1, \ldots, p_n\}, \mathrm{C})$ is used as a mark when executing methods.

The rest of this section covers the operational semantics of our language. During our commentary we give only brief descriptions of the auxiliary functions our rules reference. We refer the reader to Fig. 4 for their formal definitions.

Auxiliary Rules. [ASSIGN] and [FLD_UPDATE] (Fig. 2) are employed when executing either an assignment or field update within an SA. Each command registers the memory locations it reads and writes and stores them in its read (γ_R) and/or resp. write (γ_W) set. The read and write sets of a command aid conflict detection of atomics and gatomics (see Sects. 2.4 and 2.4). The rules [INV1], [INV2] and [INV3] facilitate method calls. [INV1] is applied when invoking a method. Note that the store s is viewed as a "stackable" mapping, where a variable p may occur several times, and $\mathsf{s}(p)$ always refers to the value of the variables p that were pushed in most recently. We use the operation push_frame(s_0, s) to "push" the frame s_0 to s, push_frame$([p \mapsto \nu], \mathsf{s})(p) = \nu$. [INV2] is applied when executing the constituent commands of a method and [INV3] is applied when a method completes. In [INV3] pop_frame(V, s) is used to "pop out" the variables

$$\frac{\delta = (\mathsf{s}_\tau \cup \sigma.\mathsf{s}, \sigma.\mathsf{h}) \quad \Gamma' = \Gamma[(\tau,\mathsf{id}) \mapsto (\Omega, \bot, \emptyset, \emptyset, \emptyset, \bot)]}{\langle \tau, \mathsf{id{:}atomic}\{c\}, \mathsf{s}_\tau, \bot\rangle, \sigma, \Gamma \xrightarrow{\mathsf{tbeg}} \langle \tau, \mathsf{id{:}ablk}(c, \mathsf{id{:}atomic}\{c\}), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma'} \ [\textsc{atomic\_beg}]$$

$$\frac{\gamma_R = \Gamma(\tau,\mathsf{id})(\mathsf{rs}) \quad \gamma_W = \Gamma(\tau,\mathsf{id})(\mathsf{ws}) \quad c, \delta, \gamma_R, \gamma_W \xrightarrow{\lambda} c', \delta', \gamma'_R, \gamma'_W \quad \Gamma' = \Gamma[(\tau,\mathsf{id}) \mapsto (\Gamma(\tau,\mathsf{id})[\mathsf{rs} \mapsto \gamma'_R, \mathsf{ws} \mapsto \gamma'_W])]}{\langle \tau, \mathsf{id{:}ablk}(c, c_1), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma \xrightarrow{\lambda} \langle \tau, \mathsf{id{:}ablk}(c', c_1), \mathsf{s}_\tau, \delta'\rangle, \sigma, \Gamma'} \ [\textsc{atomic\_update}]$$

$$\frac{\forall \tau' \in \mathcal{T}.\neg \mathsf{conflict}(\tau, \tau', \Gamma) \quad (\mathsf{s}'_\tau, \sigma') = \mathsf{merge\_upd}(\delta, \mathsf{s}_\tau, \sigma) \quad \Gamma' = \Gamma[(\tau,\mathsf{id}) \mapsto \Gamma(\tau,\mathsf{id})[\mathsf{cmt} \mapsto \Omega]]}{\langle \tau, \mathsf{id{:}ablk}(\epsilon, c_1), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma \xrightarrow{\mathsf{tcmt}} \langle \tau, \epsilon, \mathsf{s}'_\tau, \bot\rangle, \sigma', \Gamma'} \ [\textsc{atomic\_cmt}]$$

$$\frac{\forall \tau' \in \mathcal{T}.\neg \mathsf{ga\_conflict}(\tau, \tau', \Gamma') \quad \delta = (\mathsf{s}_\tau \cup \sigma.s, \sigma.h) \quad \gamma_R = \mathsf{reads}(c, \delta) \quad \gamma_W = \mathsf{writes}(c, \delta) \quad \Gamma' = \Gamma[(\tau,\mathsf{id}) \mapsto (\Omega, \bot, \gamma_R, \gamma_W, \gamma_R \cup \gamma_W, \Psi)]}{\langle \tau, \mathsf{id{:}gatomic}\{c\}, \mathsf{s}_\tau, \bot\rangle, \sigma, \Gamma \xrightarrow{\mathsf{gbeg}} \langle \tau, \mathsf{id{:}gablk}(c), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma'} \ [\textsc{gatomic\_beg}]$$

$$\frac{\exists \tau' \in \mathcal{T}.\mathsf{conflict}(\tau, \tau', \Gamma)}{\langle \tau, \mathsf{id{:}ablk}(\epsilon, c'), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma \xrightarrow{\mathsf{tabt}} \langle \tau, c', \mathsf{s}_\tau, \bot\rangle, \sigma, \Gamma \backslash \{(\tau,\mathsf{id})\}} \ [\textsc{atomic\_abt}]$$

$$\frac{c, \delta, \_, \_ \xrightarrow{\lambda} c', \delta', \_, \_}{\langle \tau, \mathsf{id{:}gablk}(c), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma \xrightarrow{\lambda} \langle \tau, \mathsf{id{:}gablk}(c'), \mathsf{s}_\tau, \delta'\rangle, \sigma, \Gamma} \ [\textsc{gatomic\_update}]$$

$$\frac{(\mathsf{s}'_\tau, \sigma') = \mathsf{merge\_upd}(\delta, \mathsf{s}_\tau, \sigma) \quad \Gamma' = \Gamma[(\tau,\mathsf{id}) \mapsto (\Gamma(\tau,\mathsf{id})[\mathsf{cmt} \mapsto \Omega])]}{\langle \tau, \mathsf{id{:}gablk}(\epsilon), \mathsf{s}_\tau, \delta\rangle, \sigma, \Gamma \xrightarrow{\mathsf{gcmt}} \langle \tau, \epsilon, \mathsf{s}'_\tau, \bot\rangle, \sigma', \Gamma'} \ [\textsc{gatomic\_cmt}]$$

$$\frac{\gamma_R = \mathsf{reads}(c, (\mathsf{s}_\tau \cup \sigma.s, \sigma.h)) \quad \gamma_W = \mathsf{writes}(c, (\mathsf{s}_\tau \cup \sigma.s, \sigma.h)) \quad \overline{\Gamma} = \Gamma[(\tau,\mathsf{id}) \mapsto (\Omega, \bot, \gamma_R, \gamma_W, \gamma_R \cup \gamma_W, \Psi)] \quad \exists \tau' \in \mathcal{T}.\mathsf{ga\_conflict}(\tau, \tau', \overline{\Gamma})}{\langle \tau, \mathsf{id{:}gatomic}\{c\}, \mathsf{s}_\tau, \bot\rangle, \sigma, \Gamma \xrightarrow{\mathsf{blk}} \langle \tau, \mathsf{id{:}gatomic}\{c\}, \mathsf{s}_\tau, \bot\rangle, \sigma, \Gamma} \ [\textsc{gatomic\_block}]$$

Fig. 3. Synchronisation Action Command Semantics

in V from the stack s, and $\mathsf{s}[p \mapsto \nu]$ changes the value of the most recent p in stack s to ν. The ϵ in [SEQ2] denotes an empty command.

Transactions [ATOMIC_BEG] (Fig. 3) is applied when an $\mathsf{atomic}\{\mathrm{C}\}$ is encountered in the source text. An atomic is relatively simple to setup: a local copy of the state, δ, is made which comprises of a store heap pair where the store entails both the thread-local and global store mappings and some default metadata is associated with the SA instance. We use Ω as a meta-timestamp that returns the current time. [ATOMIC_UPDATE] is applied for each constituent command within an atomic. When all the constituent commands of an atomic have been executed either [ATOMIC_CMT] or [ATOMIC_ABT] is applied. [ATOMIC_CMT] applies should the dataset of the atomic not conflict with any other running or

$$\mathsf{merge}(\sigma_1,\sigma_2) \stackrel{\mathsf{def}}{=} (\mathsf{mergefun}(\sigma_1.\mathsf{s},\sigma_2.\mathsf{s}),\mathsf{mergefun}(\sigma_1.\mathsf{h},\sigma_2.\mathsf{h}))$$

$$\mathsf{mergefun}(f_1,f_2)(x) \stackrel{\mathsf{def}}{=} \begin{cases} f_2(x) & x \in \mathsf{dom}(f_2) \\ f_1(x) & x \in \mathsf{dom}(f_1) \backslash \mathsf{dom}(f_2) \end{cases}$$

$$\mathsf{merge\_sets}(\sigma,\{\sigma_1,..,\sigma_n\}) \stackrel{\mathsf{def}}{=} \begin{cases} \mathsf{merge}(\sigma,\sigma_1) & n=1 \\ \mathsf{merge\_sets}(\mathsf{merge}(\sigma,\sigma_1),\{\sigma_2,..,\sigma_n\}) & n \geq 2 \end{cases}$$

$$\mathsf{merge\_upd}((\mathsf{s}_1,\mathsf{h}_1),\mathsf{s},\sigma) \stackrel{\mathsf{def}}{=} (\mathsf{mergefun}(\mathsf{s} \cup \sigma.\mathsf{s},\mathsf{s}_1),\mathsf{mergefun}(\sigma.\mathsf{h},\mathsf{h}_1))$$

$$\begin{aligned} \mathsf{conflict}(\tau_1,\tau_2,\Gamma) \stackrel{\mathsf{def}}{=} \tau_1 \neq \tau_2 \wedge \exists \mathsf{id}_1,\mathsf{id}_2 \in \mathsf{Labels}.&\Gamma(\tau_1,\mathsf{id}_1).\mathsf{ws} \cap \Gamma(\tau_2,\mathsf{id}_2).\mathsf{ds} \neq \emptyset \\ \wedge\ &(\Gamma(\tau_1,\mathsf{id}_1).\mathsf{beg} \leq \Gamma(\tau_2,\mathsf{id}_2).\mathsf{cmt} \leq \Omega \\ &\vee\ \Gamma(\tau_2,\mathsf{id}_2).\mathsf{type} = \Psi \wedge \Gamma(\tau_2,\mathsf{id}_2).\mathsf{cmt} = \bot) \end{aligned}$$

$$\begin{aligned} \mathsf{ga\_conflict}(\tau_1,\tau_2,\Gamma) \stackrel{\mathsf{def}}{=} \tau_1 \neq \tau_2 \wedge \exists \mathsf{id}_1,\mathsf{id}_2 \in \mathsf{Labels}.&\Gamma(\tau_1,\mathsf{id}_1).\mathsf{type} = \Psi \\ &\wedge \Gamma(\tau_2,\mathsf{id}_2).\mathsf{type} = \Psi \\ &\wedge \Gamma(\tau_1,\mathsf{id}_1).\mathsf{ws} \cap \Gamma(\tau_2,\mathsf{id}_2).\mathsf{ds} \neq \emptyset \wedge \Gamma(\tau_2,\mathsf{id}_2).\mathsf{cmt} = \bot \end{aligned}$$

$$\begin{aligned} \mathsf{T}_i,\sigma,\Gamma\ (\xrightarrow{\lambda_1} \circ \xrightarrow{\lambda_2})\ \mathsf{T}_i',\sigma',\Gamma' \stackrel{\mathsf{def}}{=} \exists \mathsf{T}_i'',\sigma'',\Gamma'' \cdot\ &\mathsf{T}_i,\sigma,\Gamma \xrightarrow{\lambda_1} \mathsf{T}_i'',\sigma'',\Gamma'' \\ &\wedge\ \mathsf{T}_i'',\sigma'',\Gamma'' \xrightarrow{\lambda_2} \mathsf{T}_i',\sigma',\Gamma' \end{aligned}$$

$$(\xrightarrow{\lambda})^* \stackrel{\mathsf{def}}{=} \bigcup_{r \geq 1} (\xrightarrow{\lambda})^r \qquad (\xrightarrow{\lambda})^{r+1} \stackrel{\mathsf{def}}{=} (\xrightarrow{\lambda}) \circ (\xrightarrow{\lambda})^r$$

$$\mathsf{merge\_sa}(\Gamma,\{\Gamma^1,..,\Gamma^n\}) \stackrel{\mathsf{def}}{=} \Gamma \cup \bigcup_{1 \leq i \leq n} (\Gamma^i - \Gamma)$$

Fig. 4. Auxiliary Definitions

recently committed SA. Committing an atomic entails the merging of its effect (δ) into s_τ and σ via the function merge_upd, and the updating of its SA entry. The dataset of an atomic is validated only when all its constituent commands have been executed. [ATOMIC_ABT] is applied if the atomic's dataset has been invalidated due to a conflict with another running or recently committed SA. Aborting an atomic is trivial: its SA entry is removed from Γ and the program counter of τ is set to c'. The predicate conflict in Fig. 4 determines whether or not an atomic conflicts with another SA. Informally, if a conflicting atomic or gatomic committed after or at the same time as the atomic under investigation began then the atomic is aborted.

Guaranteed Transactions [GATOMIC_BEG] is applied when gatomic{C} is encountered in the program source text. Due to the run once semantics of gatomics [GATOMIC_BEG] performs a check to see if the gatomic conflicts with any other currently running gatomic. Should a conflict exist then the gatomic cannot be immediately scheduled to run and [GATOMIC_BLOCK] is applied; otherwise, it begins execution. The functions reads and writes return the transitive closure of all locations reachable by the objects referenced within the gatomic that are read and resp. written. We resort to existing analyses (e.g., [5, 15, 20, 25])

to compute this information. The predicate ga_conflict (Fig. 4) encapsulates the pessimistic scheduling check that gatomics entail. Conceptually gatomics use two-phase locking: locks associated with the referenced objects within the gatomic are acquired before entering the gatomic and only released upon the completion of the gatomic. The temporary mapping $\overline{\Gamma}$ in [GATOMIC_BLOCK] is used to determine if the gatomic conflicts with any other running gatomic. [GATOMIC_UPDATE] is applied per each constituent command executed by the gatomic. The commands that a gatomic executes are non-instrumented versions of the commands presented in Fig. 2. [GATOMIC_CMT] is applied when the gatomic has executed all of its constituent commands.

Note that in Fig. 3, we present only semantics for atomics and gatomics, and ignore semantics for other commands (which are straightforward to define).

$$
\begin{array}{l}
[\textsc{pcomp}] \\
\textbf{1}\ \mathcal{T} = I \cup J \cup K \cup M \quad I \cup M \neq \emptyset \\
\textbf{2}\ \left\{ \begin{array}{l}
\forall i \in I \cdot \mathsf{T}_i{=}\langle i, \mathsf{id}_i{:}\mathsf{gatomic}\{c_i\}; c'_i, \mathsf{s}_i, \bot\rangle \wedge \mathsf{T}'_i{=}\langle i, c'_i, \mathsf{s}'_i, \bot\rangle \wedge \mathsf{T}''_i{=}\langle i, \mathsf{id}_i{:}\mathsf{gablk}(\epsilon); c'_i, \mathsf{s}_i, \delta_i\rangle \\
\quad \forall j \in J \cdot \mathsf{T}_j{=}\langle j, \mathsf{id}_j{:}\mathsf{gatomic}\{c_j\}; c'_j, \mathsf{s}_j, \bot\rangle \\
\quad\quad \forall k \in K \cdot \mathsf{T}_k{=}\langle k, \mathsf{id}_k{:}\mathsf{atomic}\{c_k\}; c'_k, \mathsf{s}_k, \bot\rangle \wedge \mathsf{T}''_k{=}\langle k, \mathsf{id}_k{:}\mathsf{ablk}(\epsilon, c_k); c'_k, \mathsf{s}_k, \delta_k\rangle \\
\quad\quad\quad \forall m \in M \cdot \mathsf{T}_m{=}\langle m, \mathsf{id}_m{:}\mathsf{atomic}\{c_m\}; c'_m, \mathsf{s}_m, \bot\rangle \ \wedge\ \mathsf{T}'_m = \langle m, c'_m, \mathsf{s}'_m, \bot\rangle \\
\quad\quad\quad\quad \wedge\ \mathsf{T}''_m = \langle m, \mathsf{id}_m{:}\mathsf{ablk}(\epsilon, c_m); c'_m, \mathsf{s}_m, \delta_m\rangle
\end{array} \right. \\
\textbf{3}\ \ \forall i, j \in I \cdot i{\neq}j \Rightarrow \Gamma(i, \mathsf{id}_i)(\mathsf{ws}) \cap \Gamma(j, \mathsf{id}_j)(\mathsf{ds}) = \emptyset \\
\textbf{4}\ \ \forall j \in J \cdot (\exists i \in I \cdot \Gamma(i, \mathsf{id}_i)(\mathsf{ws}) \cap \Gamma(j, \mathsf{id}_j)(\mathsf{ds}) \neq \emptyset) \\
\textbf{5}\ \left\{ \begin{array}{l}
\forall i \in I \cdot \mathsf{T}_i, \sigma, \Gamma \xrightarrow{\mathsf{gbeg}} \circ(\xrightarrow{\lambda})^* \mathsf{T}''_i, \sigma'_i, \Gamma'_i \xrightarrow{\mathsf{gcmt}} \mathsf{T}'_i, \sigma_i, \Gamma_i \ \wedge\ \Lambda_i = \mathsf{gbeg}^\frown\lambda^*{}^\frown\mathsf{gcmt} \\
\quad \forall k \in K \cdot \mathsf{T}_k, \sigma, \Gamma \xrightarrow{\mathsf{tbeg}} \circ(\xrightarrow{\lambda})^* \mathsf{T}''_k, \sigma, \Gamma'_k \xrightarrow{\mathsf{tabt}} \mathsf{T}_k, \sigma, \Gamma \\
\quad\quad \forall m \in M \cdot \mathsf{T}_m, \sigma, \Gamma \xrightarrow{\mathsf{tbeg}} \circ(\xrightarrow{\lambda})^* \mathsf{T}''_m, \sigma, \Gamma'_m \xrightarrow{\mathsf{tcmt}} \mathsf{T}'_m, \sigma_m, \Gamma_m \\
\quad\quad\quad \wedge\ \ \Lambda_m = \mathsf{tbeg}^\frown\lambda^*{}^\frown\mathsf{tcmt}
\end{array} \right. \\
\textbf{6}\ \left\{ \begin{array}{l}
\forall k \in K \cdot (\exists i \in I . \Gamma'_k(k, \mathsf{id}_k)(\mathsf{ws}) \cap \Gamma(i, \mathsf{id}_i)(\mathsf{ds}) \neq \emptyset \\
\quad \vee\ \exists m \in M . \Gamma'_k(k, \mathsf{id}_k)(\mathsf{ws}) \cap \Gamma'_m(m, \mathsf{id}_m)(\mathsf{ds}) \neq \emptyset)
\end{array} \right. \\
\textbf{7}\ \left\{ \begin{array}{l}
\forall m \in M \cdot (\forall i \in I . \Gamma'_m(m, \mathsf{id}_m)(\mathsf{ws}) \cap \Gamma(i, \mathsf{id}_i)(\mathsf{ds}) = \emptyset \ \wedge \\
\quad \forall m' \in M \backslash \{m\} . \neg\exists \mathsf{id}_{m'} . \Gamma'_m(m, \mathsf{id}_m)(\mathsf{ws}) \cap \Gamma'_{m'}(m', \mathsf{id}_{m'})(\mathsf{ds}) \neq \emptyset)
\end{array} \right. \\
\textbf{8}\ \ \sigma' = \mathsf{merge\_sets}(\sigma, \{\sigma_\tau \mid \tau \in I \cup M\}) \qquad \Gamma' = \mathsf{merge\_sa}(\Gamma, \{\Gamma_\tau \mid \tau \in I \cup M\})\ \ \textbf{9} \\
\hline
\langle \ldots || \mathsf{T}_i || .. || \mathsf{T}_j || .. || \mathsf{T}_k || .. || \mathsf{T}_m || .., \sigma, \Gamma\rangle \xrightarrow{\|_{\mathsf{mv} \in I \cup M} \Lambda_{\mathsf{mv}}} \langle .. || \mathsf{T}'_i || .. || \mathsf{T}_j || .. || \mathsf{T}_k || .. || \mathsf{T}'_m || .., \sigma', \Gamma'\rangle
\end{array}
$$

Fig. 5. Big-Step Program Move Semantics

2.5 Program Move Semantics

The orchestration of concurrently executing threads is handled by the rule [PCOMP] (Fig. 5). [PCOMP] caters for the most interesting scenario where each thread has an atomic or gatomic to run (Other scenarios are not included). It distinguishes the set of threads which make progress in their respective transition systems (*moving* threads) from those that do not (*non-moving* threads). Moving threads are executing either atomics that are to be committed or gatomics that are safe to run. To facilitate our reasoning we assume the set of concurrently executing threads $\mathcal{T}$ are split into four sets (Label 1 in Fig. 5):

- I is the set of threads that are executing code under a gatomic semantics. Every thread in I has satisfied the predicate $\neg$ga_conflict for its respective gatomic.
- J represents the set of threads that are currently blocking due to their resp. gatomics conflicting with some threads already running in I.
- M is the set of threads that can commit their atomics.
- K is the set of threads whose atomics are to be aborted.

Label 2 in Fig. 5 defines the configurations that each of the partitioned threads in I, J, M and K move through. Label 3 requires that each of the threads in I running a gatomic do not conflict w.r.t. each other. Label 4 denotes that the gatomics in threads J are currently blocking due to them conflicting with some threads currently running in I. Label 5 illustrates the transitions undertaken by each of the threads in I, J, M and K. Threads in I apply an instance of [GATOMIC_BEG], a number of [GATOMIC_UPDATE] instances and an instance of [GATOMIC_CMT]. The blocking threads in J apply an instance of [GATOMIC_BLOCK] and as such do not make any progress in their resp. transition systems. Threads in K and M differ only in their final reduction: threads in K apply instances of [ATOMIC_ABT] and those in M apply instances of [ATOMIC_CMT]. Label 6 states that the threads in K are due to abort if they conflict with either a gatomic or a committing atomic. Label 7 requires that the committing atomics do not conflict with any gatomics nor any other committing atomic. Moving threads merge their entailed effects with the program state via merge_sets (Label 8) and also merge the updates made to their respective SA entries courtesy of merge_sa (Label 9).

Each SA being executed by the threads in I, J, M and K generates a sequence of fine grained actions, λs, due to the reductions taken in their resp. transition systems. The sequence of fine grained actions generated by each SA form an action, Λ. Λ_{mv} is used to denote the actions associated with the SAs being executed by the moving threads in I and M. The fine grained actions (λs) of the actions in Λ_{mv} can be arbitrarily concurrently interleaved due to the SAs executed by threads in I and M not conflicting. This interleaving is denoted in the reduction of [PCOMP]. The resulting interleaving is governed by the same restrictions imposed by sequential consistency (SC) [17].

2.6 Properties

We show that the semantics of atomics and gatomics given in Figs. 3 and 5 are serialisable [24] and that correctly isolated programs are data-race-free.

Definition 1. *Ordered-Before ($<$). Defined over actions (Λs). Total ordering. If each fine grained action λ_i of Λ_i takes effect before the first fine grained action λ_j of Λ_j, then $\Lambda_i < \Lambda_j$.*

Intuitively if $\Lambda_i < \Lambda_j$ then we say that the effect of Λ_i *serialises-before* Λ_j. The ordered-before relation is constructed during the reduction of [PCOMP]. We show for any given execution that conflicting atomics and gatomics are serialised.

Theorem 1. *There exists a total (serialisable) order over conflicting atomics.*

Proof. Let Λ_i and Λ_j be the actions associated with the conflicting atomics T_i and resp. T_j. By definition of conflict we apply instances of [ATOMIC_CMT] and resp. [ATOMIC_ABT]. Therefore, either $\Lambda_i < \Lambda_j \vee \Lambda_j < \Lambda_i$.

Theorem 2. *There exists a total (serialisable) order over conflicting gatomics.*

Proof. Let Λ_i and Λ_j be the actions associated with the conflicting gatomics T_i and resp. T_j. By definition of ga_conflict we apply instances of [GATOMIC_BEG] and resp. [GATOMIC_BLOCK]. Therefore, either $\Lambda_i < \Lambda_j \vee \Lambda_j < \Lambda_i$.

Theorem 3. *There exists a total (serialisable) order over conflicting SAs of mixed type.*

Proof. Let Λ_i and Λ_j be the actions associated with the conflicting atomic T_i and resp. gatomic T_j. By definition of conflict we apply instances of [ATOMIC_ABT] and resp. [GATOMIC_CMT]. Therefore, $\Lambda_j < \Lambda_i$.

Intuitively gatomics have a serialisation priority over transactions due to a gatomic guaranteeing run once semantics.

A correctly isolated program is one that encapsulates every access to a memory region that is accessed by multiple threads with either an atomic or gatomic.

Theorem 4. *Correctly isolated programs are data-race-free.*

Proof. Follows from Thms. 1, 2 and 3.

3 Java Memory Model

In Sect. 2.5 we presented actions, λs, and described via [PCOMP] (Fig. 5) how these actions were permitted to be interleaved. For each execution this interleaving forms a *schedule*. In the literature a schedule is governed by a memory model [2]. Most importantly a memory model specifies the set of values a read may observe. The schedules of actions constructed in Fig. 5 were due to SC. Under SC actions from all threads appear in a totally ordered sequence, with each action respecting the program order of its issuing thread. The goal of this section is to define legal schedules of actions in terms of the Java memory model (JMM) [19]. In particular, we wish to show how *happens-before* relationships are established between atomics and gatomics. Having defined our SAs under the JMM we show that they satisfy the properties given in Sect. 2.6.

3.1 Correctly Synchronised Programs

The JMM takes a rather simple approach when defining what constitutes a correctly synchronised program, informally: any program that is *data-race-free (DRF)* [3] is guaranteed to observe an SC semantics. Before describing what it means for a program to be DRF we must cover the terminology outlined by the JMM.

- **Conflicting Accesses:** a read or write to a variable x is an *access* of x. Two accesses to x are *conflicting* if at least one of the accesses is a write.
- **Synchronisation Actions:** includes locks, unlocks, reads of volatile variables and writes to volatile variables.
- **Program Order:** the actions issued by a thread τ form a total ordering known as the program order of τ.
- **Synchronisation Order:** every execution is associated with a synchronisation order which is a *total ordering* over all SAs. Only synchronisation orders that are consistent with program order can be considered. For example, a read r of a volatile field v *must* observe the value written to v by the write w such that w occurs before r in the synchronisation order, $w \xrightarrow{so} r$. Every execution is associated with a synchronisation order.
- **Synchronises-With Order:** an unlock action a on a monitor M "synchronises-with" a *subsequent*, as defined by the synchronisation order, lock action b on M, $a \xrightarrow{sw} b$. $\xrightarrow{sw}$ is a partial order. Actions within the synchronises-with order can be issued by different threads.
- **Happens-Before Order:** is the transitive closure of the program order and synchronises-with order. An action a "happens-before" [18] another action b if a occurs before b in the happens-before ordering, written $a \xrightarrow{hb} b$.
- **Data Race:** two accesses a and b to a variable v form a data race if a and b conflict, are issued by separate threads and are not ordered by the happens-before relation.
- **Data-Race-Free Program:** a program is DRF if and only if all sequentially consistent executions of the program are free of data races.

The happens-before relation in the JMM defines the set of values a read can observe. Establishing edges in this relation requires the use of an SA. In Java such actions are generated via the use of either `volatile` variables or `synchronized` methods/blocks. As defined by the synchronises-with order there exists a pair of matching unlock and lock actions on the same monitor object M. Conceptually every monitor M is associated with an unlock action on M before any actions of a program are executed. Before we proceed further we must update the definitions of synchronisation actions and the synchronises-with order to be the following:

- **Synchronisation Actions:** includes the beginning of an atomic and gatomic (tbeg and resp. gbeg) and the end of an atomic and gatomic (tcmt and resp. gcmt).
- **Synchronises-With Order:** a tcmt (and resp. gcmt) action a on a dataset d_a "synchronises-with" a *subsequent*, as defined by the synchronisation order, tbeg (and resp. gbeg) action b on a dataset d_b when $d_a \cap d_b \neq \emptyset$, written $a \xrightarrow{sw} b$.

Our begin and end actions for atomics and gatomics are abstractions. We give their semantics in the form of the JMMs acquire and release actions in the next section.

3.2 Execution Semantics

Actions. An action $A = \langle \tau, k, v, u \rangle$ where τ is the thread performing the action, k is the kind of action being performed (discussed later), v is the variable involved in the action and $u \in \mathsf{Integers}$ is the unique identifier of the action. The kind k of an action can be one of the following: (i) write (W); (ii) read (R); (iii) acquire (Acq); or (iv) release (Rel). We do not cater for volatiles. To keep things simple we permit a write to be performed directly on a field, e.g. $\langle \tau, \mathsf{W}, v.f, 1 \rangle$. The semantics of tcmt, gbeg and gcmt are defined as follows:

$$\begin{aligned} \mathsf{tcmt} &\stackrel{\mathsf{def}}{=} \mathsf{Acq}\ \ell_i \ldots \mathsf{Acq}\ \ell_n\ \mathsf{Rel}\ \ell_n \ldots \mathsf{Rel}\ \ell_i \\ \mathsf{gbeg} &\stackrel{\mathsf{def}}{=} \mathsf{Acq}\ \ell_i \ldots \mathsf{Acq}\ \ell_n \\ \mathsf{gcmt} &\stackrel{\mathsf{def}}{=} \mathsf{Rel}\ \ell_n \ldots \mathsf{Rel}\ \ell_i \end{aligned}$$

Where each ℓ can be a variable or an object value and forms the dataset of an atomic or gatomic. We require that the contention manager has made this schedule safe and that the acquisition and release orders are topologically sorted as in McCloskey et al. [21].[1] blk is ignored (it is a sink action) and tbeg simply acts as a mark to delimit the beginning of an atomic's constituent actions. tabt does not feature in any execution as its subsequence of actions will not be observed. A valid sequence of actions for atomics and gatomics is a composition of the above with a number of reads and writes:

$$\mathsf{tbeg}\ (\mathsf{W} \mid \mathsf{R})^*\ \mathsf{tcmt} \qquad \mathsf{gbeg}\ (\mathsf{W} \mid \mathsf{R})^*\ \mathsf{gcmt}$$

Note that these sequences correspond to those of the moving threads I and M in Fig. 5.

Executions. An execution $E = \langle P, \mathsf{A}, \xrightarrow{po}, \xrightarrow{so}, \mathsf{Ws}, \mathsf{Vs}, \xrightarrow{sw}, \xrightarrow{hb} \rangle$ where P is as defined in Sect. 2, A is a set of actions, $\xrightarrow{po}$ is the program order of the actions performed by each $\tau \in \mathcal{T}$, $\xrightarrow{so}$ is the synchronisation order, Ws is a write-seen function, Vs is a value-seen function and $\xrightarrow{sw}$ and $\xrightarrow{hb}$ are as defined previously.

4 Related Work

Shavit and Touitou [27] introduced software transactional memory (STM). The isolation semantics afforded by an STM are either weak or strong [6, 7, 11]. Literature on the semantics of STM includes that by Abadi et al. [1] which is based on the automatic mutual exclusion (AME) [14] concurrent programming language. Koskinen et al. [16] have also studied the semantics of STM but their work does not entail the mixing of pessimistic and optimistic concurrency control.

[1] This model is used only to illustrate a projection onto the JMMs existing SAs.

Ziarek et al. [32] described a dynamic approach for selecting a stronger semantics when an atomic attempted to execute an operation which seems (determined by a magic analysis) to require stronger guarantees than that afforded by an atomic. Unfortunately, such a semantics reverts to using programmer specified lock invariants which are error prone. Smaragdakis et al. [28] presented a set of language extensions to temporarily "suspend" an atomic's isolation in order to support irreversible operations, however they rely heavily on the specification of isolation invariants, which are again, error prone. Privatisation and publication [30] can be used to emulate a stronger semantics within STM but requires the programmer to correctly transfer ownership of memory regions between threads.

Ni et al. [23] championed *obstinate* transactions but are a product of a prior abort. Welc et al. [31] use single owner read locks to transition to a guaranteed semantics but permit only a single such atomic to run at any given time. Sonmez et al. [29] present a model built on Haskell STM that turns atomics that access "hot" regions of memory into pessimistic atomics, however this approach again is dynamic and does not afford dataset guarantees. Autolocker [21] presents a model of pessimistic atomics by using a type system that uses programmer specified lock protection annotations to convert atomics into lock-based equivalents statically. Recent literature such as that by McCloskey et al. [21], Ni et al. [23], Shavit and Matveev [26] and Welc et al. [31] have, via empirical evidence, more than justified not only the practical feasibility of pessimistic concurrency control for STM but also its importance in simplifying the programming model.

Adding atomics to a language such as Java impacts the underlying memory model [2] as outlined by Ziarek et al. [32] and Grossman et al. [9]. Menon et al. [22] provide a number of properties that memory models must take into consideration for supporting atomics, such as the Java memory model (JMM) [19].

5 Summary

We have presented a small-step operational semantics for a programming language that supports compositional mixed mode concurrency control for transactional programs. Our language partitions transactions into two types: atomics (optimistic) and gatomics (pessimistic). Gatomics guarantee run once semantics and the safe use of the privatisation and publication idioms. We also showed that the reads and writes issued by atomics and gatomics are serialisable under both a sequential consistency and Java memory model semantics.

Acknowledgment. Granville Barnett is supported by EPSRC Doctoral Training Award. Shengchao Qin is supported in part by EPSRC project EP/G042322.

References

[1] Abadi, M., Birrell, A., Harris, T., Isard, M.: Semantics of transactional memory and automatic mutual exclusion. In: Principles of Programming Languages (2008)
[2] Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. Computer (1996)

[3] Adve, S.V., Hill, M.D.: Weak ordering – a new definition. In: International Symposium on Computer Architecture (1990)
[4] Agrawal, R., Carey, M.J., Livny, M.: Concurrency control performance modeling: alternatives and implications. ACM Trans. Database Syst. (1987)
[5] Andersen, L.O.: Program analysis and specialization for the c programming language. Technical report. University of Copenhagen (1994)
[6] Blundell, C., Christopher Lewis, E., Martin, M.M.K.: Deconstructing transactional semantics: The subtleties of atomicity. In: Workshop on Duplicating, Deconstructing, and Debunking (2005)
[7] Blundell, C., Devietti, J., Christopher Lewis, E., Martin, M.M.K.: Making the fast case common and the uncommon case simple in unbounded transactional memory. In: International Symposium on Computer Architecture (2007)
[8] Gray, J.: The transaction concept: virtues and limitations. Very Large Data Bases (1981)
[9] Grossman, D., Manson, J., Pugh, W.: What do high-level memory models mean for transactions? In: Memory System Performance and Correctness (2006)
[10] Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Principles and Practice of Parallel Programming (2005)
[11] Harris, T., Larus, J., Rajwar, R.: Transactional memory, 2nd edn. (2010)
[12] Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Principles of Distributed Computing (2003)
[13] Hickey, R.: The clojure programming language. In: Dynamic Languages Symposium (2008)
[14] Isard, M., Birrell, A.: Automatic mutual exclusion. In: USENIX (2007)
[15] Jenista, J., Demsky, B.: Disjointness analysis for java-like languages. Technical report, University of California, Irvine (2009)
[16] Koskinen, E., Parkinson, M., Herlihy, M.: Coarse-grained transactions. In: Principles of Programming Languages (2010)
[17] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. Transactions on Computers (1979)
[18] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM (1978)
[19] Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: Principles of Programming Languages (2005)
[20] Marron, M., Méndez-Lojo, M., Hermenegildo, M., Stefanovic, D., Kapur, D.: Sharing analysis of arrays, collections, and recursive structures. In: Program Analysis for Software Tools and Engineering (2008)
[21] McCloskey, B., Zhou, F., Gay, D., Brewer, E.: Autolocker: synchronization inference for atomic sections. In: Principles of Programming Languages (2006)
[22] Menon, V., Balensiefer, S., Shpeisman, T., Adl-Tabatabai, A.-R., Hudson, R.L., Saha, B., Welc, A.: Practical weak-atomicity semantics for java stm. In: Symposium on Parallelism in Algorithms and Architectures (2008)
[23] Ni, Y., Welc, A., Adl-Tabatabai, A.-R., Bach, M., Berkowits, S., Cownie, J., Geva, R., Kozhukow, S., Narayanaswamy, R., Olivier, J., Preis, S., Saha, B., Tal, A., Tian, X.: Design and implementation of transactional constructs for c/c++. In: Object-Oriented Programming Systems Languages and Applications (2008)
[24] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM (1979)
[25] Pugh, W., Wonnacott, D.: Constraint-based array dependence analysis. Transactions on Programming Languages and Systems (1998)

[26] Shavit, N., Matveev, A.: Towards a fully pessimistic stm model. Transactional Computing (2012)
[27] Shavit, N., Touitou, D.: Software transactional memory. In: Principles of Distributed Computing (1995)
[28] Smaragdakis, Y., Kay, A., Behrends, R., Young, M.: Transactions with isolation and cooperation. In: Object-Oriented Programming Systems Languages and Applications (2007)
[29] Sonmez, N., Harris, T., Cristal, A., Unsal, O.S., Valero, M.: Taking the heat off transactions: Dynamic selection of pessimistic concurrency control. In: International Symposium on Parallel and Distributed Processing (2009)
[30] Spear, M.F., Marathe, V.J., Daless, L., Scott, M.L.: Privatization techniques for software transactional memory. In: Principles of Distributed Computing (2007)
[31] Welc, A., Saha, B., Adl-Tabatabai, A.-R.: Irrevocable transactions and their applications. In: Symposium on Parallelism in Algorithms and Architectures (2008)
[32] Ziarek, L., Welc, A., Adl-Tabatabai, A.-R., Menon, V., Shpeisman, T., Jagannathan, S.: A Uniform Transactional Execution Environment for Java. In: Dell'Acqua, P. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 129–154. Springer, Heidelberg (2008)

Towards a Formal Verification Methodology for Collective Robotic Systems*

Edmond Gjondrekaj[1], Michele Loreti[1], Rosario Pugliese[1], Francesco Tiezzi[2], Carlo Pinciroli[3], Manuele Brambilla[3], Mauro Birattari[3], and Marco Dorigo[3]

[1] Dipartimento di Sistemi e Informatica, Università degli Studi di Firenze, Italy
[2] IMT, Institute for Advanced Studies Lucca, Italy
[3] IRIDIA, CoDE, Université Libre de Bruxelles, Belgium

Abstract. We present a novel formal verification approach for *collective robotic systems* that is based on the use of the formal language KLAIM and related analysis tools. While existing approaches focus on either micro- or macroscopic views of a system, we model aspects of both the robot hardware and behaviour, as well as relevant aspects of the environment. We illustrate our approach through a robotics scenario, in which three robots cooperate in a decentralized fashion to transport an object to a goal area. We first model the scenario in KLAIM. Subsequently, we introduce random aspects to the model by stochastically specifying actions execution time. Unlike other approaches, the specification thus obtained enables quantitative analysis of crucial properties of the system. We validate our approach by comparing the results with those obtained through physics-based simulations.

1 Introduction

Collective robotic systems are systems in which a group of autonomous robots cooperates to tackle a task. By taking advantage of the absence of a centralized controller, the use of local communication and sensing, and the lack of global knowledge, these systems have the potential to display properties such as robustness and parallelism.

Collective robotic systems are difficult to design and analyse because the collective behaviour of the system is the result of the non-linear interaction of the individual robots with each other and with the environment. The realisation of these systems currently relies on the ingenuity and expertise of the designer due to the lack of sound engineering approaches and accountable engineering practices. The typical design approach involves several loops of development, testing and modification of the behaviour of each robot until the desired collective behaviour is obtained. This iterative process, often performed first using computer simulations and eventually on real robots, is in general expensive, time consuming, and cannot provide guarantees of system correctness. Indeed, *experimentation* with real robots is very costly and time consuming. Physics-based *simulation*, that attempts to realistically model the environment, the robots and their interactions, is faster and more reliable than experimentation, but requires an exhaustive scan of the design parameter space to reach any conclusion (see e.g. [18]). Besides, experimentation and simulation can only validate a small subset of the possible

* This work has been partially sponsored by the EU project ASCENS, 257414.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 54–70, 2012.

system scenarios and are often impractical to exhaustively study a collective behaviour. In other words, these approaches cannot ensure a complete coverage of the critical aspects of the system nor the absence of residual anomalies.

A major open issue in the design and development of collective robotic systems is thus to guarantee the correctness of the collective behaviour of a system composed of autonomous components. Formal verification techniques, such as *model checking*, can complement traditional approaches by guaranteing that certain system properties hold.

In this paper, we introduce a formal verification approach for the design of collective robotic systems that lays down the basis of a principled development methodology for such systems. Our approach consists of two phases. In the first phase, we model the robot behaviour and the environment with the formal language KLAIM [7]. KLAIM is a tuple-space-based coordination language that allows one to define an accurate model of a distributed system using a small set of primitives. Unlike existing approaches that mainly focus on micro- or macroscopic aspects of the system, KLAIM permits to capture both hardware aspects of the robots and their behaviour. In addition, KLAIM can suitably model relevant environmental aspects. In the second phase, we enrich the model with stochastic aspects, using KLAIM's stochastic extension STOKLAIM [8], and formalise the desired properties using MOSL [8]. MOSL is a stochastic logic that, in addition to qualitative properties, permits specifying time-bounded probabilistic reachability properties and properties about resource distribution. The properties of interest are then verified against the STOKLAIM specifications by exploiting the analysis tool SAM [8,19].

To demonstrate the approach, we analyse a collective transport scenario [10], in which, while avoiding obstacles, a group of three robots must carry an object that is too heavy for a single robot to move. This behaviour is a good candidate to establish the validity of our approach since it has many of the features which characterize collective robotic systems. Indeed, the system is completely distributed (there is neither a centralized controller nor a leader), the robots do not have any global knowledge, such as a map of the environment, and no common frame of reference is used for coordination.

Modelling collective robotic systems is a challenging task. Indeed, for understanding their dynamics, it is necessary to model in detail both the spatial aspects (e.g. positions of robots, obstacles and carried objects) and the temporal aspects (e.g. robots' action execution time) of a system. These aspects are crucial since, e.g., without the position of robots and carried objects in time the model would be of limited use to verify the correctness of the collective transport behaviour. Differently from the relevant literature, in which spatial and temporal aspects are usually discarded or simplified (see, e.g., [18,12]), our approach allows us to easily achieve the needed level of detail. The price to be paid is an increased complexity of the model that might limit the number of robots that can be considered in a given scenario.

In addition to suit collective robotic systems, compositionality and high modularity, which are typical of KLAIM and STOKLAIM specifications, allow us to easily and flexibly experiment with different values of the behaviour and scenario parameters. This permits tuning them in order, for example, to optimise the performance of the system or prevent instabilities. Moreover, the possibility to change the parameters of the environment permits to easily check the collective behaviour of the robots under different environmental conditions while saving time and resources with respect to simulated or

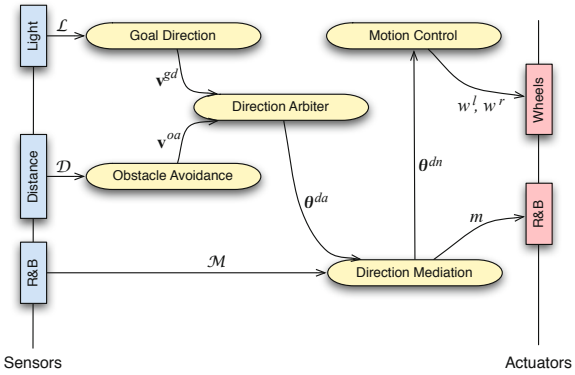
real robots. E.g., regarding our scenario, we have studied how robots' behaviour is influenced by the position of the light source indicating the goal area. We have verified that if the robots do not perceive the light they could not be able to reach the goal, and demonstrated that a simple change to their behaviour is enough to solve this problem.

State of the art and related work. Formal verification has been successfully applied to many different classes of distributed systems, such as embedded real-time systems and wireless sensor networks [6,20]. These kinds of systems, even though distributed, do not present the challenges of robotic systems. In fact, their components do not move and do not interact with the environment as a robot does. Considering more specifically modelling and verification of collective robotic systems, most of the work focuses on systems that are not fully distributed as they have a centralized controller or a leader, or make use of global knowledge. Examples of such work can be found in [15,11].

Only a few studies deal explicitly with robotic systems which are fully distributed and do not use global knowledge. Winfield et al. [21] devised a microscopic modelling approach based on linear temporal logic (LTL) to model a swarm of robots whose goal is to navigate in the environment while keeping in communication range. The same approach was then studied and expanded in [9]. Konur et al. [17] proposed to use probabilistic computation tree logic (PCTL) to formally verify the properties of a swarm of robots that perform a foraging task using a macroscopic model. A similar property-driven approach is proposed in [3], where PRISM is used to verify PCTL formulae expressing properties of an aggregation scenario, described by a Discrete Time Markov Chain, where the robots have to cluster in an area of the environment. The design methodology proposed in [16] exploits a *post hoc* analysis to evaluate the expected performance of synthesized robot controllers. Such analysis does not permit verifying generic system properties, but just determining the probability of correct task execution to refine the controller synthesis. Moreover, robot systems are not specified through a linguistic approach, but in terms of states, functions over states, and state transitions. In [4], the use of Maude and related tools is put forward for analysing a self-assembling robots scenario, where robots physically connect to each other when the environment prevents them from reaching their goals individually. The work focusses on the adaptive aspects of the system, while abstracting from the spatial one, since the arena is modelled as a discrete grid and robots movements are discretized into four directions.

All the above approaches greatly simplify the spatial and/or temporal aspects of the system and are thus not suited for a collective transport behaviour. Moreover, in contrast to these approaches, we focus on analysing the properties of a robot's behaviour both from the point of view of the interaction between the running code and the robot's internal devices (i.e. sensors and actuators) and from the point of view of the interaction with the other robots. To the best of our knowledge, there are no published works that deal with modelling and formal verification of a collective transport behaviour.

Summary of the rest of the paper. In Section 2, we introduce the considered robotics scenario. In Section 3, we review the formal basis underlying the proposed verification approach, namely the specification language KLAIM, the stochastic extension STOKLAIM, the stochastic logic MoSL, and the analysis tool SAM. In Section 4, we describe the relevant aspects of the KLAIM specification of the scenario, while in Section 5, we present its stochastic analysis. Finally, in Section 6 we indicate directions for future work.



Symbol	Meaning
$\mathcal{L}$	Set of light readings
$\mathcal{D}$	Set of distances
$\mathcal{M}$	Set of received messages
m	Sent message
$\mathbf{v}^{gd}$	Vector toward to the goal area
$\mathbf{v}^{oa}$	Vector to avoid obstacles
θ^{da}	Individual desired direction
w^l/w^r	Left/right wheel speed
θ^{dn}	Mediated direction

Fig. 1. A diagrammatic description of the behaviour for collective transport of an object

2 A Collective Robotics Scenario

In order to illustrate our approach, we consider a robotics scenario, borrowed from [10], whereby three identical robots must collectively transport an object to a goal area. The robots operate in an arena where a number of obstacles are present and a light source indicates the goal area. It is assumed that the three robots have already physically assembled to the object to transport and cannot disassemble until the goal area is reached.

Each robot is a marXbot [2] equipped with: (i) a *light sensor*, to perceive the direction to a light source; (ii) a *distance scanner*, to obtain relative distances from objects in the environment; (iii) a *range and bearing communication system*, to communicate with other robots; (iv) *wheels*, to move around the environment.

All robots execute the same code, i.e., the so-called *behaviour*. Each of them senses the environment and calculates the *desired direction*, that is, the direction the robot would follow if it were alone. Since each robot has a local perception of the environment, the desired directions of the robots could differ. In fact, at each control step, one robot could sense or not the position of the goal and/or the position of obstacles. According to the available information, in different moments, a robot can be *informed*, that is, it has a desired direction to follow, or *non-informed* otherwise. Informed robots communicate to the other robots their desired direction. Anyway, to actuate the wheels, any robot uses a *socially mediated direction* obtained by averaging the received directions, so all robots can follow the same direction even if they have a different perception of the environment.

A diagrammatic description of the behaviour, together with an explanation of the used notation, is reported in Fig. 1. The horizontal blobs are behavioural modules that take an input and produce an output. The output is usually a set of variables that can be input to other modules or set as actuator values, while the input can be the result of other modules and/or sensor readings. The behaviour is composed of five modules.

Goal Direction queries the light sensors to calculate the vector $\mathbf{v}^{gd}$ to the position of maximum light intensity sensed, which points toward to the goal area. 24 light sensors are located around the body of the robot in a ring at uniform fixed angles, and each

of them returns the measured intensity expressed as a vector directed from the robot's center outwards. The vector $\mathbf{v}^{gd}$ is calculated as a normalised sum over these 24 vectors. *Obstacle Avoidance* detects the presence of obstacles and calculates the vector $\mathbf{v}^{oa}$ that points away from the obstacle. The distance scanner is a rotating sensor that can span the area around the robot and return 24 vectors whose length corresponds to the distance to a sensed object from the center of the robot (if no obstacle is perceived along a given direction, the length of the vector is obs_d_{MAX}). The length of vector $\mathbf{v}^{oa}$ corresponds to the distance to the closest object rescaled in [0, 1], while its angle corresponds to the angle of the sum of all the readings. Notably, the resulting angle points away from the closest obstacles, because the readings that correspond to obstacle-free areas have the highest value obs_d_{MAX}. *Direction Arbiter* takes as inputs $\mathbf{v}^{gd}$ and $\mathbf{v}^{oa}$ and calculates the direction θ^{da}, that is the desired direction of the robot before computing the mediated direction. Since the length of $\mathbf{v}^{oa}$ represents how urgent it is to avoid obstacles, we use it as a weight to combine the directions to the goal area and to avoid obstacles. *Direction Mediation* calculates the mediated direction θ^{dn} as the average of the directions received from other robots through the range and bearing communication system. This module sends a message m to nearby robots containing θ^{da}, if the robot is informed, and θ^{dn} otherwise. *Motion Control* converts the direction θ^{dn} into the wheel speeds w^r and w^l.

3 Formal Foundations of the Verification Approach

In this section, we provide a brief overview of the formal methods exploited by the proposed approach for specifying and verifying collective robotic systems.

Specification. A distributed system is modelled in KLAIM as a net of nodes, each one with a local data repository and a set of running processes. We informally present here a version of KLAIM enriched with some standard control flow constructs (i.e., if-then-else, sequence, etc.) that are part of the input language of the analysis tool used in Section 5. These constructs simplify the specification task and can be easily rendered in the language originally presented in [7]. For simplicity's sake, we omit the linguistic constructs for dealing with name restriction and dynamic node creation, since they are not used in the considered robotics system specification. We refer to [7] for a formal presentation of the language and to [1] for a Java framework for programming in KLAIM.

Nets are finite plain collections of nodes composed by means of the parallel composition operator _ || _. *Nodes* $s ::_\rho C$ have a unique *locality name* s (i.e., their network address) and an allocation environment ρ, and host a set of components C. An *allocation environment* provides a name resolution mechanism by mapping *locality variables* l (i.e., aliases for addresses), occurring in the processes hosted in the corresponding node, into localities s. The distinguished locality variable **self** is used by processes to refer to the address of their current hosting node. *Components* are finite plain collections of processes P and evaluated tuples $\langle t \rangle$, composed by means of the parallel operator _ | _.

Processes P are the KLAIM active computational units and may be executed concurrently either at the same locality or at different localities. They are built up from basic actions (see below) and process calls $A(p_1, \ldots, p_m)$ by means of sequential composition $P_1; P_2$, parallel composition $P_1 \mid P_2$, conditional choice **if** (e) **then** $\{P\}$ **else** $\{Q\}$, iterative constructs **for** $i = n$ **to** m { P } and **while** (e) $\{P\}$, and (possibly recursive) process

definitions $A(f_1, \ldots, f_n) \triangleq P$ with f_i pairwise distinct. Notably, A denotes a process identifier, while f_i and p_j denote formal and actual parameters, respectively. Moreover, e ranges over *expressions*, which contain basic values (booleans, integers, strings, floats, etc.) and value variables x, and are formed by using the standard operators on basic values, simple data structures (i.e., arrays and lists) and the non-blocking retrieval actions **inp** and **readp** (explained below). In the rest of this section, we will use the notation ℓ to range over locality names and locality variables.

During their execution, processes perform some *basic actions*. Actions $\mathbf{in}(T)@\ell$ and $\mathbf{read}(T)@\ell$ are retrieval actions and permit to withdraw/read data tuples from the tuple space hosted at the (possibly remote) locality ℓ: if a matching tuple is found, one is non-deterministically chosen, otherwise the process is blocked. They exploit templates as patterns to select tuples in shared tuple spaces. *Tuples* t are sequences of actual fields, i.e. locality names, locality variables, expressions and processes. Instead, *templates* T are sequences of actual and formal fields, where the latter are written $!\,x$, $!\,l$ or $!\,X$ and are used to bind variables to values, locality names or processes, respectively. For the sake of readability, we use "_" to denote a *don't care* formal field in a template; this corresponds to a formal field $!\,dc$ using the variable dc that does not occur elsewhere in the specification. Actions $\mathbf{inp}(T)@\ell$ and $\mathbf{readp}(T)@\ell$ are non-blocking versions of the retrieval actions: namely, during their execution processes are never blocked. Indeed, if a matching tuple is found, **inp** and **readp** act similarly to **in** and **read**, and additionally return the value *true*; otherwise they return the value *false* and the executing process does not block. $\mathbf{inp}(T)@\ell$ and $\mathbf{readp}(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is simply ignored). Action $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space of the target node identified by ℓ, while action $\mathbf{eval}(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Both **out** and **eval** are non-blocking actions. Action $\mathbf{rpl}(T) \rightarrow (t)@\ell$ atomically replaces a non-deterministically chosen tuple in ℓ matching the template T by the tuple t; if no tuple in ℓ matches T, the action behaves as $\mathbf{out}(t)@\ell$. Finally, action $x := e$ assigns the value of e to x and, differently from all the other actions, it is not indexed with an address because it always acts locally.

Verification. Quantitative analysis of a KLAIM specification can be enabled by associating a rate to each action, thus obtaining a STOKLAIM [8] specification. This rate is the parameter of an exponentially distributed random variable accounting for the action duration time. A real valued random variable X has a *negative exponential distribution* with *rate* $\lambda > 0$ if and only if the *probability* that $X \leq t$, with $t > 0$, is $1 - e^{-\lambda \cdot t}$. The expected value of X is λ^{-1}, while its variance is λ^{-2}. The operational semantics of STOKLAIM permits associating to each specification a Continuous Time Markov Chain that can be used to perform quantitative analyses of the considered system.

The desired properties of a system under verification are formalised using the stochastic logic MoSL [8]. MoSL formulae use predicates on the tuples located in the considered KLAIM net to express the reachability of the system goal, or more generally, of a certain system state, while passing or not through other specific intermediate states. Therefore, MoSL can be used to express quantitative properties of the overall system behaviour, such as, e.g., if the robots are able to reach the goal, or collisions between the robots and the obstacles ever happen in the system. The results of the evaluation of

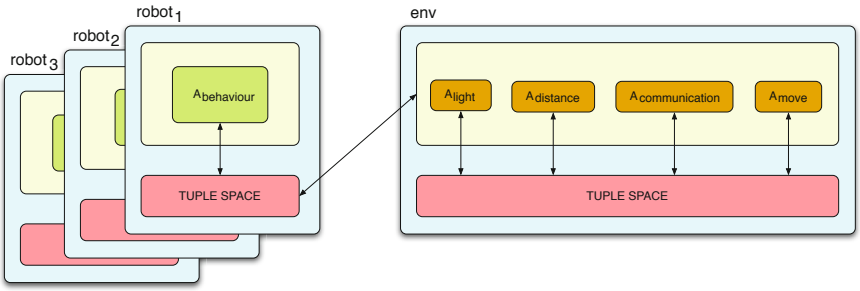


Fig. 2. Graphical representation of the KLAIM specification

such properties do not have a rigid meaning, like *true* or *false*, but have a less absolute nature, e.g. *in 99.7% of the cases, the robots reach the goal within t time units*.

Verification of MoSL formulae over STOKLAIM specifications is assisted by the analysis tool SAM [8,19], which uses a statistical model checking algorithm [5] to estimate the probability of the property satisfaction. In this way, the probability associated to a path-formula is determined after a set of independent observations and the algorithm guarantees that the difference between the computed value and the exact one exceeds a given *tolerance* ε with a probability that is less than a given *error probability* p.

4 Specification of the Robotics Scenario

In this section, we present the KLAIM specification of the robots' behaviour informally introduced in Section 2. Moreover, to formally analyse the behaviour, we specify the low-level details about the robots and the arena where the robots move, i.e., the obstacles, the goal, etc. We use KLAIM to model also these aspects because, on the one hand, the language is expressive enough to suitably represent them and, on the other hand, this approach enables the use of existing tools for the analysis of KLAIM specifications.

Here, we focus only on the *qualitative* aspects of the scenario. In the next section, our specification will be enriched with *quantitative* aspects by simply associating a rate to each KLAIM action, thus obtaining a STOKLAIM specification.

The scenario model. The overall scenario is rendered in KLAIM by the following net

$$
\begin{aligned}
& robot_1 ::_{\{\mathbf{self} \mapsto robot_1\}} A_{behaviour} \mid C_{robotData\,1} \\
\parallel\ & robot_2 ::_{\{\mathbf{self} \mapsto robot_2\}} A_{behaviour} \mid C_{robotData\,2} \\
\parallel\ & robot_3 ::_{\{\mathbf{self} \mapsto robot_3\}} A_{behaviour} \mid C_{robotData\,3} \\
\parallel\ & env ::_{\{\mathbf{self} \mapsto env, r_1 \mapsto robot_1, r_2 \mapsto robot_2, r_3 \mapsto robot_3\}} A_{light} \mid A_{distance} \mid A_{communication} \mid A_{move} \mid C_{envData}
\end{aligned}
$$

which is graphically depicted in Fig. 2. The three robots are modelled as three KLAIM nodes whose locality names are $robot_1$, $robot_2$ and $robot_3$. Similarly, the environment around the robots is rendered as a node, with locality name *env*, as well. The allocation environment of each robot node contains only the binding for **self** (i.e., $\mathbf{self} \mapsto robot_i$), while the allocation environment of the *env* node contains the binding for **self** (i.e., $\mathbf{self} \mapsto env$) and the bindings for the robot nodes (i.e., $r_i \mapsto robot_i$, with $i \in \{1, 2, 3\}$).

The behaviour is rendered as a process identified by $A_{behaviour}$, which is exactly the same in all three robots. The items of *local knowledge* data $C_{robotData\,i}$ of each robot are

stored in the tuple space of the corresponding node and consist of sensor readings and computed data; at the outset, such data are the sensor readings at the initial position.

The processes running on the *env* node provide environmental data to the robots' sensors and keep this information up-to-date as time goes by according to the actions performed by the robots' actuators. The process A_{light}, given the position of the light source and the current position of the robots, periodically computes the information about the light position for each robot and sends it to them. This data corresponds to the values obtained from light sensors and is stored in the tuple space of each robot. Similarly, the process $A_{distance}$ provides each robot with information about the obstacles around it. The process $A_{communication}$ models the communication infrastructure and, hence, takes care of delivering the messages sent by the robots by means of their range and bearing communication systems. Finally, the process A_{move} periodically updates the robots' positions according to their directions.

The data within the *env* node can be *static*, such as the information about the obstacles and the source of light, or *dynamic*, such as the robots' positions. The tuples $C_{envData}$ are stored in the tuple space of this node and their meaning is as follows (as usual, strings are enclosed within double quotes): $\langle$"*pos*", $x_1, y_1, x_2, y_2, x_3, y_3\rangle$ represents the positions (x_1, y_1), (x_2, y_2) and (x_3, y_3) of the three robots; $\langle$"*light*", $x_l, y_l, i\rangle$ represents a light source, with intensity i and origin in (x_l, y_l), indicating the goal position; $\langle$"*obstacles*", $m\rangle$ indicates the total number of obstacles present in the environment (this permits simplifying the scanning of obstacles data in the KLAIM specification); and $\langle$"*obs*", $n, x_1, y_1, x_2, y_2, x_3, y_3, x_4, y_4\rangle$ represents the n-th rectangular-shaped obstacle, with vertices (x_1, y_1), (x_2, y_2), (x_3, y_3) and (x_4, y_4).

It is worth noticing that, while the KLAIM process $A_{behaviour}$ is intended to model the actual robot's behaviour (e.g., it could be used as a skeleton to generate the code of the behaviour), the KLAIM processes and data representing the robots' running environment (i.e., sensors, actuators, obstacles, goal, etc.) are just models of the environment and of physical devices, which are needed to enable the analysis.

The robot model. Each robot executes a behaviour that interacts with the robot's tuple space for reading and producing sensors and actuators data to cyclically perform the following activities: *sensing* data about the local environment, *elaborating* the retrieved knowledge data to make decisions, and *acting* according to the elaborated decisions (i.e., it transmits data to other robots and actuates the wheels to perform a movement).

Different choices can be made when developing the model of the robot [18]. We have chosen to model individually the behaviour of each robot and the corresponding sensors and actuators. We illustrate in this section the data associated to the robots' sensors and actuators, and the KLAIM specification of the robots' behaviour (due to lack of space, the rest of the specification is relegated to a companion technical report [14]).

Robots' sensor and actuator data. The *light sensor* data is rendered in KLAIM as a tuple of the form $\langle$"*light*", $\boldsymbol{\ell}\rangle$, where "*light*" is a *tag* indicating the sensor originating the data while $\boldsymbol{\ell}$ is an array of 24 elements. For each $i \in [0, 23]$, $\boldsymbol{\ell}[i]$ represents the light intensity perceived by the sensor along the direction $2\pi\frac{i}{24}$.

The tuple containing the measures of the *distance scanner sensor* is similar. Indeed, it is of the form $\langle$"*obs*", $\mathbf{d}\rangle$, where "*obs*" is the tag associated to *distance scanner sensor*

data and **d** is an array of 24 elements. For each $i \in [0, 23]$, $\mathbf{d}[i]$ is the distance to the closest obstacle measured by the sensor along the direction $2\pi\frac{i}{24}$.

The *range and bearing communication system* acts as both a sensor and an actuator. Indeed, it allows a robot to send messages to other robots in its neighborhood and to receive messages sent by them. A process running in the environment node is used to read (and consume) the messages produced by each robot's behaviour and to *route* them to the other robots (through the environment node). This process models the communication medium and specifies the range and bearing communication system without considering explicitly the details of the underlying communication framework. Each robot stores received messages in a local tuple of the form $\langle$"*msgs*", $[m_1, m_2, \ldots, m_n]\rangle$ representing a queue of length n containing messages m_1,m_2,...,m_n[1]. Instead, to send a message to other robots, a behaviour locally stores a tuple of the form $\langle$"*msg*", $m\rangle$. The process running on the environment node is in charge of reading each message and propagating it to the other robots that are in the sender's communication range.

Finally, the *wheel actuators* are rendered as a process running in the environment node that reads the new directions to be followed by the robots (i.e., tuples of the form $\langle$"*move*", $\theta\rangle$) and updates the robots' position (which is, in fact, an information stored in the tuple space of the environment node). This slightly differs from the original specification given in Section 2, where the *Motion Control* module converts the direction calculated by the *Direction Mediation* module into speeds for the two wheels, which are then passed to the wheels actuator. In fact, although our specification is quite detailed, it is still an abstract description of a real-world scenario. Thus, some details that do not affect the analysis, such as those involving the calculation of the robots' movements, are considered at an higher level of abstraction.

For simplicity's sake, we do not consider noise and failures of sensors and actuators.

Robot's behaviour. The process $A_{behaviour}$, modelling the robot's behaviour graphically depicted in Fig. 1, is defined as follows:

$$A_{behaviour} \triangleq A_{goalDirection} \mid A_{obstacleAvoidance} \mid A_{directionArbiter} \mid A_{directionMediation} \mid A_{motionControl}$$

Each behavioural module (i.e., a yellow blob in Fig. 1) corresponds to one of the above KLAIM processes, whose definitions are provided below. The specification code is made self-explanatory through the use of comments (i.e. strings starting with //).

The *Goal Direction* module takes as input the last light sensors readings (here rendered as a tuple of the form $\langle$"*light*", $\boldsymbol{\ell}\rangle$) and returns the vector $\mathbf{v}^{gd}$ (actually, here only the direction of $\mathbf{v}^{gd}$ is returned, because its length is always 1 and does not play any role on the computation of the new direction to be followed). This behavioural module is modelled by the recursive process $A_{goalDirection}$ defined as follows:

$A_{goalDirection} \triangleq$
x_{sum} , y_{sum} := 0;
read("*light*", !ℓ)@**self**; // read the tuple containing the light sensor readings
for i = 0 **to** 23{
  x_{sum} := $x_{sum} + \ell[i] \cdot \cos(2\pi i/24)$; // calculate the coordinates of the final point of the...
  y_{sum} := $y_{sum} + \ell[i] \cdot \sin(2\pi i/24)$; // ... vector (with the origin as initial point) resulting...
}; // ... from the vectorial sum of the reading vectors

[1] $[v_1, \ldots, v_n]$ denotes a list of n elements, [] the empty list, and :: the concatenation operator.

if $((x_{sum} \,!=\, 0) \wedge (y_{sum} \,!=\, 0))$ **then** { // check if the light is perceived
  $\angle \mathbf{v}^{gd} := Angle(0, 0, x_{sum}, y_{sum})$; // calculate $\angle \mathbf{v}^{gd}$, i.e., the direction of vector $\mathbf{v}^{gd}$
  rpl("vgd", _) $\rightarrow$ ("vgd", $\angle \mathbf{v}^{gd}$)@**self**; // update the vector $\mathbf{v}^{gd}$ data
} **else** {
  inp("vgd", _)@**self** // if the light is not preceived, remove the previous vector $\mathbf{v}^{gd}$ data
}; $A_{goalDirection}$

The sensor readings are always present in the tuple space, because they are present at the outset and the processes modelling behavioural modules do not consume sensor data while reading or updating them. Therefore, the **read** action before the **for** loop above never blocks the execution of process $A_{goalDirection}$. In principle, by pooling the tuple space in this way, the same sensor data could be read more than once; this faithfully reflects the actual interaction model between the robots code and the sensors. Anyway, it does not lead to divergent behaviours during the analysis, because such interactions are regulated by the action rates specified in the StoKlaim model (see Section 5).

The function $Angle(x_0, y_0, x_1, y_1)$, used above and in subsequent parts of the specification, returns the direction (i.e., the angle) of the vector from (x_0, y_0) to (x_1, y_1). We refer the interested reader to [13] for its definition.

The *Obstacle Avoidance* module takes as input the last distance sensors readings (here rendered as a tuple of the form $\langle$"obs", $\mathbf{d}\rangle$) and returns the vector $\mathbf{v}^{oa}$. This behavioural module is modelled by the process $A_{obstacleAvoidance}$ defined as follows:

$A_{obstacleAvoidance} \triangleq$
x_{sum} , $y_{sum} := 0$; $min := obs\_d_{MAX}$;
read("obs", !d)@**self**; // read the tuple representing the distance sensor readings
for $i = 0$ **to** 23{
  $x_{sum} := x_{sum} + d[i] \cdot \cos(2\pi i/24)$; // calculate the coordinates of the final point of the...
  $y_{sum} := y_{sum} + d[i] \cdot \sin(2\pi i/24)$; // ... vectorial sum of the reading vectors
  if $(d[i] < min)$ **then** $min := d[i]$ // calculate the minimum length of the vectors
};
$\| \mathbf{v}^{oa} \| := min/obs\_d_{MAX}$; // calculate $\| \mathbf{v}^{oa} \|$, i.e., the length of vector $\mathbf{v}^{oa}$ rescaled in $[0, 1]$
$\angle \mathbf{v}^{oa} := Angle(0, 0, x_{sum}, y_{sum})$; // calculate $\angle \mathbf{v}^{oa}$, i.e., the direction of vector $\mathbf{v}^{oa}$
rpl("voa", _, _) $\rightarrow$ ("voa", $\| \mathbf{v}^{oa} \|, \angle \mathbf{v}^{oa}$)@**self**; // update the vector $\mathbf{v}^{oa}$ data
$A_{obstacleAvoidance}$

where obs_d_{MAX} is the maximum range of the distance sensor (in [10], it is set to 1.5 m).

The process $A_{directionArbiter}$ modelling the *Direction Arbiter* module, which takes $\mathbf{v}^{gd}$ and $\mathbf{v}^{oa}$ as input and returns the direction θ^{da}, is defined as follows:

$A_{directionArbiter} \triangleq$
in("voa", !voa_l, !θ^{oa})@**self**; // read and consume the tuple containing $\mathbf{v}^{oa}$ (it's always present)
if (**inp**("vgd", !θ^{gd})@**self**) **then** { // read and consume the tuple containing $\mathbf{v}^{gd}$ (if available)
  $v_x^{da} := (1 - voa\_l) \cdot cos(\theta^{oa}) + voa\_l \cdot cos(\theta^{gd})$; // calculate the coordinates of the...
  $v_y^{da} := (1 - voa\_l) \cdot sin(\theta^{oa}) + voa\_l \cdot sin(\theta^{gd})$; // ... vector to the desired direction
  $\theta^{da} := Angle(0, 0, v_x^{da}, v_y^{da})$; // compute the angle θ^{da}
  rpl("da", _) $\rightarrow$ ("da", θ^{da})@**self**; // update the angle θ^{da} data
} **else** {
  if $(voa\_l < 1)$ **then** { // check if any obstacle has been detected
    rpl("da", _) $\rightarrow$ ("da", θ^{oa})@**self** // use the obstacle avoidance direction as θ^{da}
  }
}; $A_{directionArbiter}$

Notably, differently from sensor readings, data produced by other modules (e.g. $\mathbf{v}^{gd}$ and $\mathbf{v}^{oa}$) are removed from the tuple space when read.

The *Direction Mediation* module takes as input the direction θ^{da} computed by the *Direction Arbiter* and the last received messages (here rendered as a tuple of the form $\langle$"*msgs*", $[m_1, m_2, \ldots, m_n]\rangle$) and returns the direction θ^{dn}, to be used by the *Motion Control* module, and a message m, to be sent to the other robots via the range and bearing system. The *Direction Mediation* module is modelled by the following process:

$A_{directionMediation} \triangleq$
$c\,,\ sum_x\,,\ sum_y\ :=\ 0;$
rpl("*msgs*", $!l$) $\rightarrow$ ("*msgs*", [])@**self**; // read and reset the list of received messages
while ($l == \theta :: tail$) { // scan the list
  $l\ :=\ tail$;
  $sum_x\ :=\ sum_x + cos(\theta)$; // calculate the sum of the received...
  $sum_y\ :=\ sum_y + sin(\theta)$; // ... directions
  $c\ :=\ c + 1$ // increase the counter of the received messages
};
if ($c == 0$) **then** { // check if there are received messages
  if (**inp**("*da*", $!\theta^{da}$)@**self**) **then** { // if there aren't, check if the robot is informed
    rpl("*dir*", _) $\rightarrow$ ("*dir*", θ^{da})@**self**; // update the direction data for the motion control
    rpl("*msg*", _) $\rightarrow$ ("*msg*", θ^{da})@**self** // update the message to be sent to the other robots
  }
} **else** { // if there are received messages,...
  $\theta^{dn}\ :=\ Angle(\ 0,\ 0,\ sum_x,\ sum_y\ )$; // ... calculate the average direction and proceed
  (**rpl**("*dir*", _) $\rightarrow$ ("*dir*", θ^{dn})@**self** // update the data for the motion control
    |
    (**if** (**inp**("*da*", $!\theta^{da}$)@**self**) **then** { // check if the robot is informed
        $m\ :=\ \theta^{da}$ // if it is, the produced message contains θ^{da}
      } **else** {
        $m\ :=\ \theta^{dn}$ // if the robot is not informed, the produced message contains θ^{dn}
      };
      rpl("*msg*", _) $\rightarrow$ ("*msg*", m)@**self** // update the message to be sent to the other robots
)
)
}; $A_{directionMediation}$

Notice that the tuple containing the direction θ^{da} is consumed when read. Thus, to avoid blocking the execution of the process, to read such tuple an action **inp** (within the condition of an **if** construct) is exploited.

The *Motion Control* module takes as input the direction computed by the *Direction Mediation* module and transmits it to the wheels actuator. The *Motion Control* module is modelled by the following process:

$A_{motionControl} \triangleq$
in("*dir*", $!\theta^{dn}$)@**self**; // wait (and consume) a direction of movement
rpl("*move*", _) $\rightarrow$ ("*move*", θ^{dn})@**self**; // transmit the direction to the wheels actuator
$A_{motionControl}$

As previously explained, we do not need to model the conversion of the direction calculated by the *Direction Mediation* module into speeds for the wheels.

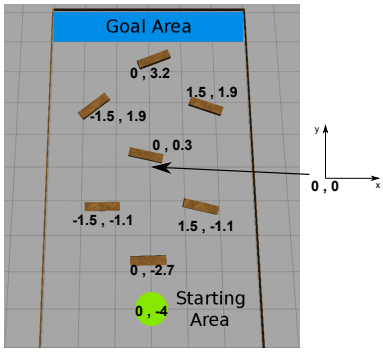


Fig. 3. Arena and initial configuration

5 Stochastic Specification and Analysis

In this section, we demonstrate how the KLAIM specification presented n the previous section can support the analysis. The proposed methodology can be used to verify the system success, obtaining accurate estimations of the system performance expressed in terms of the probability of reaching the goal without entering unwanted intermediate states. This permits making comparisons of the algorithm performance in different scenarios, which may use different features of the obstacles, transported objects, terrain, and also different goals and requirements. It also permits to analyse different details of the system behaviour, which can provide helpful information for understanding the reasons why an unwanted behaviour is observed and can allow the system designer to tune the system in order to improve its performance under different conditions. Our approach relies on formal tools, like stochastic modal logics and model checking, that permits expressing and evaluating performance measures in terms of logical formulae. In this way, we obtain a framework for the analysis of collective robotic systems which is more abstract and expressive than existing simulation frameworks, where the analysis is typically performed by relying on an *a posteriori* data analysis. Moreover, for the sake of efficiency, simulators are usually deterministic, e.g. all robots act synchronously. This means that some possible behaviours of a real system are not taken into account in the simulation. Instead, stochastic modelling tools permit considering the typical *uncertainty* of real systems, e.g. by abstracting from the precise scheduling of robot movements. In this way, developers are guaranteed that their analyses cover more critical situations of the considered scenario, according to a given margin of error.

We now enrich the KLAIM specification introduced in the previous section with stochastic aspects and consider the scenario configuration presented in [10] and depicted in Fig. 3. Seven rectangular objects are scattered in the arena, while the light source is positioned high above the goal area and is always visible to the robots. We assume that robots, on average, are able to perform 10 sensor readings per second and that they have an average speed of $2cm/sec$, and let the part of the specification modeling the environment be able to perform a mean of 100 operations per second. Starting from these parameters we have derived specific rates for defining the STOKLAIM specification.

As an excerpt of the StoKlaim specification, we report below the stochastic definition of process $A_{obstacleAvoidance}$:

$A_{obstacleAvoidance} \triangleq$
$x_{sum}\ ,\ y_{sum}\ :=\ 0;\ \ min\ :=\ obs\_d_{MAX};$
read("*obs*", !*d*)@**self** : λ_1 ;
for $i = 0$ **to** 23{ ... }; $\|\ \mathbf{v}^{oa}\ \| :=\ min/obs\_d_{MAX};\ \angle\mathbf{v}^{oa}\ :=\ Angle(0, 0, x_{sum}, y_{sum});$
rpl("*voa*", _, _) → ("*voa*", $\|\ \mathbf{v}^{oa}\ \|, \angle\mathbf{v}^{oa}$)@**self** : λ_2 ;
$A_{obstacleAvoidance}$

The actions highlighted by a gray background are those annotated with rates λ, where $\lambda_1 = 24.0$ and $\lambda_2 = 90.0$. These rates guarantee that obstacle avoidance data are updated every $\frac{1}{24} + \frac{1}{90}$ time units on average, i.e. about 20 times per second. We refer the interested reader to [13] for the rest of the stochastic specification.

The result of a simulation run of the StoKlaim specification, performed by using Sam, is reported in Fig. 4 (a). The trajectories followed by the three robots in this run are plotted in the figure with three different colors; they show that the robots reach the goal without collisions. On an Apple iMac computer (2.33 GHz Intel Core 2 Duo and 2 GB of memory) simulation of a single run needs an average time of 123 seconds.

We have analysed the probability to reach the goal without colliding with any obstacles. The property "*robots have reached the goal area*" is formalized in MoSL, for the specific system under analysis, by the formula ϕ_{goal} defined below:

$$\phi_{goal} = \langle \text{“}pos\text{”}, !x_1, !y_1, !x_2, !y_2, !x_3, !y_3 \rangle @ env \rightarrow y_1 \geq 4.0 \wedge y_2 \geq 4.0 \wedge y_3 \geq 4.0$$

This formula relies on *consumption* operator, $\langle T \rangle @ l \rightarrow \phi$, that is satisfied whenever a tuple matching template T is located at l and the remaining part of the system satisfies ϕ. Hence, formula ϕ_{goal} is satisfied if and only if tuple $\langle \text{“}pos\text{”}, x_1, y_1, x_2, y_2, x_3, y_3 \rangle$, where each y_i is greater than 4.0, is in the tuple space located at *env* (all robots are in the goal area). Similarly, the property "*a robot collided an obstacle*" is formalized by:

$$\phi_{col} = \langle \text{“}collision\text{”} \rangle @ env \rightarrow \mathsf{true}$$

where tuple $\langle$"*collision*"$\rangle$ is located at *env* whenever a robot collided an obstacle.

The considered analyses have been then performed by estimating the total probability of the set of runs satisfying $\neg\phi_{col} U^{\leq t} \phi_{goal}$ where the formula $\phi_1 U^{\leq t} \phi_2$ is satisfied by all the runs that reach within t time units a state satisfying ϕ_2 while only traversing states that satisfy ϕ_1. In the analysis, a time-out of $500sec$ has been considered.

Under the configuration of Fig. 3, i.e. when the robots are always able to perceive the light, we get that the goal can be reached without collisions with probability 0.916, while robots do not reach the goal or collide with obstacles with probability 0.084 (these values have been estimated with parameters $p = 0.1$ and $\varepsilon = 0.1$, 1198 runs). Such results are in accordance with those reported in [10], where the estimated probability to reach the goal is 0.94. The slight variation is mainly due to a different way of rendering robots movement, which is computed via a *physical simulator* in [10], while in our case it is approximated as the vectorial sum of the movement of each single robot.

We have then modified the original scenario by locating the light source on the same plane of the arena and we noticed that the overall system performances are deeply influenced. Indeed, since objects cast shadows, they can prevent robots from sensing

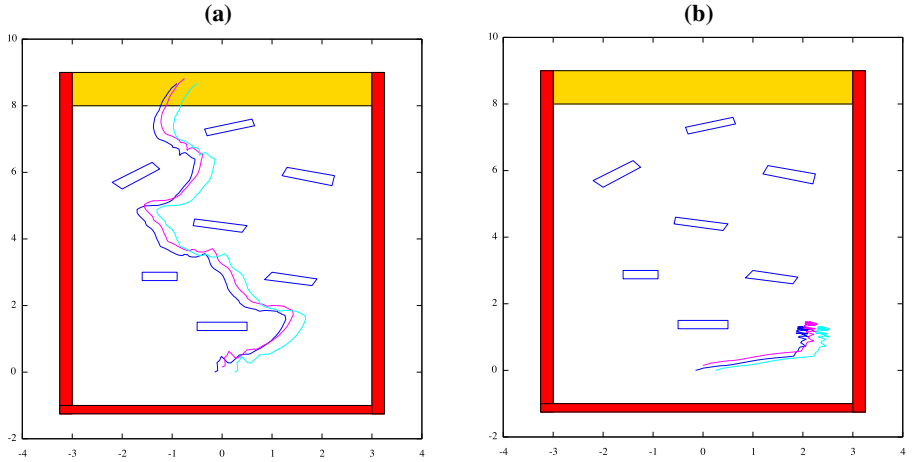


Fig. 4. Some simulation results obtained for the robotics scenario from [10]

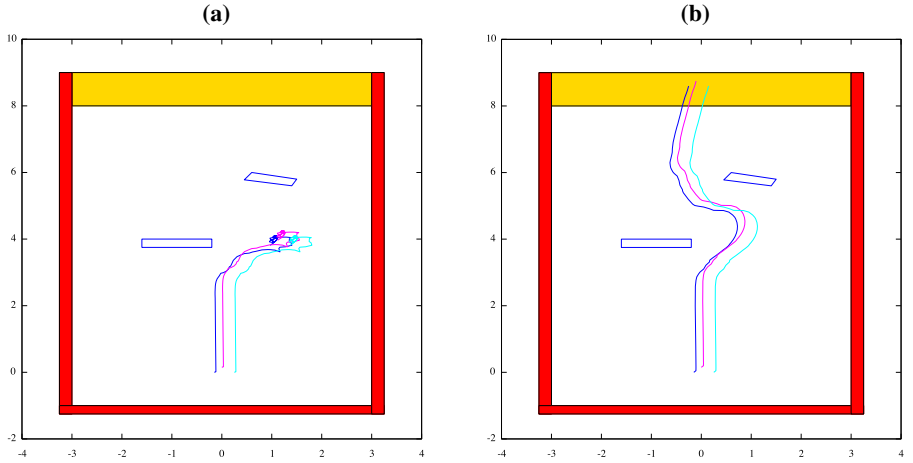


Fig. 5. Some simulation results obtained for a simple robotics scenario

the light. Under this configuration, the robots are not able to reach the goal area (see Fig. 4 (b) for a simulation trace representing a sort of counterexample for the given property) and the probability to reach the goal without collisions plummets to 0.0.

In order to validate our model and to verify the robots' behaviour, we have also considered other scenarios. In Fig. 5 (a) we show a simpler scenario where just two obstacles are placed at the center of the arena. At the beginning, the obstacles do not hide the light to the robots. However, when the first obstacle enters in the range of the robots' distance sensors, the robots turn to right, enter in the shadow cast by the second object and then never reach the goal area. This problem can be avoided by modifying the robot behaviour so that, when the light is not perceived, the last known goal direction is used. The adoption of this simple policy increases the probability to reach the goal area without collisions, from 0.234 to 1.0 (see the simulation run in Fig. 5 (b)).

6 Concluding Remarks

We have presented a novel approach to formal verification of collective robotic systems and validated it against a traditional approach consisting in physics-based simulations. We have shown that the obtained results are in accordance with those resulting from physics-based simulations and reported in [10], which have been in fact exploited for tuning the quantitative aspects of our analysis (e.g. the robots' actions execution time).

Our approach paves the way for the definition of a 5-step engineering methodology based on formal foundations. In the first step, the designer models the system formally with KLAIM. In the second step, he adds stochastic aspects to enable its analysis and analyses the system properties to discover flaws in the formal model. These two steps can be iterated, allowing the designer to discover and fix flaws of the system even before the actual code is written. In the third step, the specification is converted into (the skeleton of) the robots behaviour code. In the fourth step, the code is tested with physics-based simulations, to reveal further model-worthy aspects that were neglected in the first two steps. Finally, in the fifth step, robots behaviour is tested on real robots. The focus of this paper is on the definition of the first two steps.

We believe that the development methodology we envisage has many advantages when compared with ad-hoc design and validation through physics-based simulation and experimentation with real robots. Indeed, it permits to formally specify the requirements of the system and to minimize the risk of developing a system that does not satisfy the required properties, as these properties are checked at each step of the development phase. It also permits detecting potential design flaws early in the development process thus reducing the cost of later validation efforts.

Depending on the complexity of the system to develop, implementing the model for enabling physics-based simulation might not be straightforward and could require the ingenuity and expertise of the developer. Therefore, we intend to define and implement an automatic translation from KLAIM specifications of robot behaviours to actual code that can be taken as input by the physics-based simulator. This would allows us to complete the 5-step development process mentioned above. We also plan to apply our approach to other challenging robotic scenarios, by studying the performance of different robot behaviours while changing environmental conditions and system requirements.

Moreover, we intend to consider more abstract system specifications to conveniently deal with swarm robotics scenarios. In fact, to enable the verification of the class of properties we deemed interesting for the collective robotics domain, we have defined a very detailed model of the system under analysis. Indeed, the model we propose permits taking into account, during the analysis process, the exact position of each robot, as well as any other information about its internal state, at each instant of time. The model fits well with collective transport scenarios, where usually a limited number of robots are involved; however, it may become not tractable using available tools when the number of robots significantly grows. To deal with such kind of scenarios, like e.g. the swarm robotics one, the abstraction level of the model has to be gradually raised up in accordance with an increasing number of robots, by focussing on those aspects of the system that become most relevant. This approach would be reasonable in case of

swarms, because the properties of interest are no longer related to the exact position of each single robots, but concern the global (abstract) behaviour of the overall system.

References

1. Bettini, L., De Nicola, R., Pugliese, R.: Klava: a Java Package for Distributed and Mobile Applications. Software - Practice and Experience 32(14), 1365–1394 (2002)
2. Bonani, M., et al.: The marxbot, a miniature mobile robot opening new perspectives for the collective-robotic research. In: IROS, pp. 4187–4193. IEEE (2010)
3. Brambilla, M., Pinciroli, C., Birattari, M., Dorigo, M.: Property-driven design for swarm robotics. In: AAMAS. IFAAMAS (to appear, 2012)
4. Bruni, R., Corradini, A., Gadducci, F., Lluch Lafuente, A., Vandin, A.: Modelling and Analyzing Adaptive Self-assembly Strategies with Maude. In: Durán, F. (ed.) WRLA 2012. LNCS, vol. 7571, pp. 118–138. Springer, Heidelberg (2012)
5. Calzolai, F., Loreti, M.: Simulation and Analysis of Distributed Systems in KLAIM. In: Clarke, D., Agha, G. (eds.) COORDINATION 2010. LNCS, vol. 6116, pp. 122–136. Springer, Heidelberg (2010)
6. Clarke, E.M., Wing, J.M.: Formal methods: state of the art and future directions. ACM Comput. Surv. 28, 626–643 (1996)
7. De Nicola, R., Ferrari, G., Pugliese, R.: KLAIM: A Kernel Language for Agents Interaction and Mobility. Transactions on Software Engineering 24(5), 315–330 (1998)
8. De Nicola, R., Katoen, J., Latella, D., Loreti, M., Massink, M.: Model checking mobile stochastic logic. Theor. Comput. Sci. 382(1), 42–70 (2007)
9. Dixon, C., Winfield, A., Fisher, M.: Towards Temporal Verification of Emergent Behaviours in Swarm Robotic Systems. In: Groß, R., Alboul, L., Melhuish, C., Witkowski, M., Prescott, T.J., Penders, J. (eds.) TAROS 2011. LNCS, vol. 6856, pp. 336–347. Springer, Heidelberg (2011)
10. Ferrante, E., Brambilla, M., Birattari, M., Dorigo, M.: Socially-Mediated Negotiation for Obstacle Avoidance in Collective Transport. In: Martinoli, A., Mondada, F., Correll, N., Mermoud, G., Egerstedt, M., Hsieh, M.A., Parker, L.E., Støy, K. (eds.) Distributed Autonomous Robotic Systems. STAR, vol. 83, pp. 571–583. Springer, Heidelberg (2013)
11. Fisher, M., Wooldridge, M.: On the formal specification and verification of multi-agent systems. Int. Journal of Cooperative Information Systems 6(1), 37–66 (1997)
12. Galstyan, A., Hogg, T., Lerman, K.: Modeling and Mathematical Analysis of Swarms of Microscopic Robots. In: SIS, pp. 201–208. IEEE (2005)
13. Gjondrekaj, E., Loreti, M., Pugliese, R., Tiezzi, F.: Specification and Analysis of a Collective Robotics Scenario in SAM (2011), SAM source file available at http://rap.dsi.unifi.it/SAM/
14. Gjondrekaj, E., et al.: Towards a formal verification methodology for collective robotic systems. Technical report, Univ. Firenze (2011), http://rap.dsi.unifi.it/~loreti/papers/collective_transport_verification.pdf
15. Jeyaraman, S., et al.: Formal techniques for the modelling and validation of a co-operating UAV team that uses Dubins set for path planning. In: ACC, vol. 7, pp. 4690–4695. IEEE (2005)
16. Jones, C., Mataric, M.J.: Synthesis and Analysis of Non-Reactive Controllers for Multi-Robot Sequential Task Domains. In: Ang, M.H., Khatib, O. (eds.) Experimental Robotics IX. STAR, vol. 21, pp. 417–426. Springer, Heidelberg (2004)

17. Konur, S., Dixon, C., Fisher, M.: Formal Verification of Probabilistic Swarm Behaviours. In: Dorigo, M., Birattari, M., Di Caro, G.A., Doursat, R., Engelbrecht, A.P., Floreano, D., Gambardella, L.M., Groß, R., Şahin, E., Sayama, H., Stützle, T. (eds.) ANTS 2010. LNCS, vol. 6234, pp. 440–447. Springer, Heidelberg (2010)
18. Lerman, K., Martinoli, A., Galstyan, A.: A Review of Probabilistic Macroscopic Models for Swarm Robotic Systems. In: Şahin, E., Spears, W.M. (eds.) Swarm Robotics WS 2004. LNCS, vol. 3342, pp. 143–152. Springer, Heidelberg (2005)
19. Loreti, M.: SAM: Stochastic Analyser for Mobility, http://rap.dsi.unifi.it/SAM/
20. Stankovic, J.A.: Strategic directions in real-time and embedded systems. ACM Comput. Surv. 28, 751–763 (1996)
21. Winfield, A., et al.: On Formal Specification of Emergent Behaviours in Swarm Robotic Systems. Int. Journal of Advanced Robotic Systems 2(4), 363–370 (2005)

Modeling Resource-Aware Virtualized Applications for the Cloud in Real-Time ABS*

Einar Broch Johnsen, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa

Department of Informatics, University of Oslo, Norway
{einarj,rudi,sltarifa}@ifi.uio.no

Abstract. An application's quality of service (QoS) depends on resource availability; e.g., response time is worse on a slow machine. On the cloud, a virtualized application leases resources which are made available on demand. When its work load increases, the application must decide whether to reduce QoS or increase cost. Virtualized applications need to manage their acquisition of resources. In this paper resource provisioning is integrated in high-level models of virtualized applications. We develop a Real-Time ABS model of a cloud provider which leases virtual machines to an application on demand. A case study of the Montage system then demonstrates how to use such a model to compare resource management strategies for virtualized software during software design. Real-Time ABS is a timed abstract behavioral specification language targeting distributed object-oriented systems, in which dynamic deployment scenarios can be expressed in executable models.

1 Introduction

The added value and compelling business drivers of cloud computing are undeniable [10], but considerable new challenges need to be addressed for industry to make an effective usage of cloud computing. As the key technology enabler for cloud computing, *virtualization* makes elastic amounts of resources available to application-level services deployed on the cloud; for example, the processing capacity allocated to a service may be changed on the demand. The integration of virtualization in general purpose software applications requires novel techniques for leveraging resources and resource management into software engineering. Virtualization poses challenges for the software-as-a-service abstraction concerning the development, analysis, and dynamic composition of software with respect to quality of service. Today these challenges are not satisfactorily addressed in software engineering. In particular, better support for the modeling and validation of application-level resource management strategies for virtualized resources are needed to help the software developer make efficient use of the available virtualized resources in their applications.

* Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu).

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 71–86, 2012.

The abstract behavioral specification language ABS is a formalism which aims at describing systems at a level which abstracts from many implementation details but captures essential behavioral aspects of the targeted systems [25]. ABS targets the engineering of concurrent, component-based systems by means of executable object-oriented models which are easy to understand for the software developer and allow rapid prototyping and analysis. The extension Real-Time ABS integrates object orientation and timed behavior [8]. Whereas the functional correctness of a planned system largely depends on its high-level behavioral specification, the choice of deployment architecture may hugely influence the system's quality of service. For example, CPU limitations may restrict the applications that can be supported on a cell phone, and the capacity of a server may influence the response time of a service during peaks in the user traffic.

Whereas software components reflect the logical architecture of systems, *deployment components* have recently been proposed for Real-Time ABS to reflect the deployment architecture of systems [27,28]. A deployment component is a resource-restricted execution context for a set of concurrent object groups, which controls how much computation can occur in this set between observable points in time. Deployment components may be dynamically created and are parametric in the amount of resources they provide to their objects. This explicit representation of deployment scenarios allows application-level response time and load balancing to be expressed in the software models in a very natural and flexible way, relative to the resources allocated to the software.

This paper shows how deployment components in Real-Time ABS may be used to model virtualized systems in a cloud environment. We develop a Real-Time ABS model of cloud provisioning and accounting for resource-aware applications: an abstract cloud provider offers virtual machines with given CPU capacities to client applications and bills the applications for their resource usage. We use this model in a case study of the Montage system [24], a cloud-based resource-aware application for scientific computing, and compare execution times and accumulated costs depending on the number of leased machines by means of simulations of the executable model. We show that our results are comparable to those previously obtained for Montage with the same deployment scenarios on specialized simulation tools [19] and thus that our formal model can be used to estimate cloud deployment costs for realistic systems. We then introduce dynamic resource management strategies in the Montage model, and show that these improve on the resource management strategies previously considered [19].

The paper is structured as follows. Section 2 presents the abstract behavioral specification language Real-Time ABS, Section 3 develops our model of cloud provisioning. Section 4 presents the case study of the Montage system. Section 5 discusses related work and Section 6 concludes the paper.

2 Abstract Behavioral Specification with Real-Time ABS

ABS is an executable object-oriented modeling language with a formal semantics [25], which targets distributed systems. The language is based on concurrent

object groups, akin to concurrent objects (e.g., [14,17,26]), Actors (e.g., [1,23]), and Erlang processes [5]. Concurrent object groups in ABS internally support interleaved concurrency using guarded commands. This allows active and reactive behavior to be easily combined, based on cooperative scheduling of processes which stem from method calls. A concurrent object group has at most one active process at any time and a queue of suspended processes waiting to execute on an object in the group. Objects in ABS are dynamically created from classes but typed by interface; i.e., there is no explicit notion of hiding as the object state is always encapsulated behind interfaces which offer methods to the environment.

2.1 Modeling Timed Behavior in ABS

ABS combines functional and imperative programming styles with a Java-like syntax [25]. Concurrent object groups execute in parallel and communicate through asynchronous method calls. Data manipulation inside methods is modeled using a simple functional language. Thus, the modeler may abstract from the details of low-level imperative implementations of data structures, and still maintain an overall object-oriented design which is close to the target system.

The *functional* part of ABS allows user-defined algebraic data types such as the empty type `Unit`, Booleans `Bool`, integers `Int`; parametric data types such as sets `Set<A>` and maps `Map<A>` (given a value for the variable `A`); and user-defined functions over values of these types, with support for pattern matching.

The *imperative* part of ABS addresses concurrency, communication, and synchronization at the concurrent object level, and defines interfaces, classes, and methods. ABS objects are *active* in the sense that their `run` method, if defined, gets called upon creation. *Statements* for sequential composition $s_1; s_2$, assignment `x=rhs`, **`skip`**, **`if`**, **`while`**, and **`return`** are standard. The statement **`suspend`** unconditionally suspends the active process of an object by moving this process to the queue, from which an enabled process is selected for execution. In **`await`** g, the guard g controls suspension of the active process and consists of Boolean conditions b and return tests x? (see below). Functional expressions e and guards g are side-effect free. If g evaluates to false, the active process is suspended, i.e., moved to the queue, and some process from the queue may execute. *Expressions* `rhs` include the creation of an object group **`new cog`** `C(e)`, object creation in the creator's group **`new`** `C(e)`, method calls `o!m(e)` and `o.m(e)`, future dereferencing `x.get`, and functional expressions `e`.

Communication and *synchronization* are decoupled in ABS, which allows complex workflows to be modeled. Communication is based on asynchronous method calls, denoted by assignments `f=o!m(e)` where `f` is a future variable, `o` an object expression, and `e` are (data value or object) expressions. After calling `f=o!m(e)`, the future variable `f` refers to the return value of the call and the caller may proceed with its execution *without blocking* on the method reply. There are two operations on future variables, which control synchronization in ABS. First, the statement **`await`** `f?` *suspends the active process* unless a return value from the call associated with `f` has arrived, allowing other processes in the object group to execute. Second, the return value is retrieved by the expression

`f.get`, which *blocks all execution in the object* until the return value is available. The statement sequence `x=o!m(e);v=x.get` encodes commonly used *blocking calls*, abbreviated `v=o.m(e)` (reminiscent of synchronous calls).

We work with Real-Time ABS [8], a timed extension of ABS with a run-to-completion semantics, which combines *explicit* and *implicit* time for ABS models. Real-Time ABS has an interpreter defined in rewriting logic [30] which closely reflects its semantics and which executes on the Maude platform [16]. In Real-Time ABS, explicit time is specified directly in terms of durations (as in, e.g., UPPAAL [29]). Real-Time ABS provides the statement **`duration`**`(b,w)` to specify a duration between the worst-case `w` and the best case `b`. A process may also suspend for a certain duration, expressed by **`await duration`**`(b,w)`. For the purposes of this paper, it is sufficient to work with a discrete time domain, and let `b` and `w` be of type `Int`. In contrast to explicit time, implicit time is *observed* by measurements of the executing model. Measurements are obtained by comparing clock values from a global clock, which can be read by an expression **`now`**`()` of type `Time`. With implicit time, no assumptions about execution times are hard-coded into the models. The execution time of a method call depends on how quickly the call is effectuated by the server object. In fact, the execution time of a statement varies with the *capacity* of the chosen deployment architecture and on *synchronization* with other (slower) objects.

2.2 Modeling Deployment Architectures in Real-Time ABS

Deployment components in Real-Time ABS abstractly capture the resource capacity at a location [27,28]. Deployment components are first-class citizens in Real-Time ABS and share their resources between their allocated objects. The root object of a model is allocated to the deployment component `environment`, which has unlimited resources. Deployment components with different resource capacities may be dynamically created depending on the control flow of the model or statically created in the main block of the model. When created, objects are by default allocated to the same deployment component as their creator, but they may also be explicitly allocated to a different component by an annotation.

Deployment components have the type `DC` and are instances of the class `DeploymentComponent`. This class takes as parameters a name (the name of the location, mostly used for monitoring purposes), given as a string, and a set of restrictions on resources. Here we focus on resources reflecting the components' *CPU processing* capacity, which are specified by the constructor `CPUCapacity(r)`, where `r` of type `Resource` represents the amount of available abstract processing resources between observable points in time. The expression **`thisDC`**`()` evaluates to the deployment component of the current object. The method `total("CPU")` of a deployment component returns the total amount of CPU resources allocated to that component.

The *CPU processing capacity* of a deployment component determines how much computation may occur in the objects allocated to that component. The CPU resources of a component define its capacity between observable (discrete) points in time, after which the resources are renewed. Objects allocated to the

component compete for the shared resources in order to execute. With the run-to-completion semantics, the objects may execute until the component runs out of resources or they are otherwise blocked, after which time will advance [28].

The *cost* of executing statements is given by a cost model. A default cost value for statements can be set as a compiler option (e.g., `defaultcost=10`). This default cost does not discriminate between different statements. For some statements a more precise cost expression is desirable in a realistic model; e.g., if `e` is a complex expression, then the statement `x=e` should have a significantly higher cost than the statement **`skip`**. For this reason, more fine-grained costs can be introduced into the models by means of annotations, as follows:

```
class C implements I {
  Int m (T x) { [Cost: g(size(x))] return f(x); }
}
```

It is the responsibility of the modeler to specify an appropriate cost model. A behavioral model with default costs may be gradually refined to obtain more realistic resource-sensitive behavior. To provide cost functions such as `g` in our example above, the modeler may be assisted by the COSTABS tool [2], which computes a worst-case approximation of the cost for `f` in terms of the size of the input value `x` based on static analysis techniques, when given the definition of the expression `f`. However, the modeler may also want to capture resource consumption at a more abstract level during the early stages of system design, for example to make resource limitations explicit before further behavioral refinements of a model. Therefore, cost annotations may be used to abstractly represent the cost of some computation which remains to be fully specified.

3 Resource Management and Cloud Provisioning

An explicit model of cloud provisioning allows the application developer to interact in a simple way with a provisioning and accounting system for virtual machines. This section explains how such cloud provisioning may be modeled, for Infrastructure-as-a-Service [10] cloud environments. Consider an interface `CloudProvider` which offers three methods for resource management to client applications: `createMachine`, `acquireMachine`, and `releaseMachine`.

The method `createMachine` prepares and returns an abstract virtual machine with a specified processing capacity, after which the client application may deploy objects on the machine. This method models the provisioning and configuration part of a cloud-based application, and corresponds roughly to instancing and configuring a virtual machine on a cloud, without starting up the machine.

Before running a computation on a machine created with `createMachine`, the client application must first call the method `acquireMachine`. The cloud provider then starts *accounting* for the time this machine is kept running; the client calls the method `releaseMachine` to "shut down" the machine again. (For simplicity it is currently not checked whether processes are run before calling `acquireMachine` or after `releaseMachine`; this is a straightforward extension of the approach which could be useful to model "cheating" clients.)

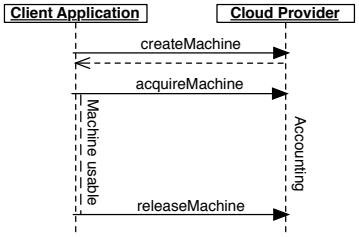


Fig. 1. Interaction between a client application and the cloud provider

For a later reactivation of the same machine, only `acquireMachine` needs to be called. Fig. 1 shows one such sequence of interactions between a client application and a cloud provider.

In addition, the interface offers a method `getAccumulatedCost` which returns the cost accumulated so far by the client application. This method can be used in load balancing schemes to implement various trade-offs between quality of service and the cost of running the application, or to implement operator alerts when certain QoS or cost budgets are bypassed.

A Model of Cloud Provisioning in Real-Time ABS. A class which implements the `CloudProvider` interface is given in Fig. 2. Abstract virtual machines are modeled as deployment components. The class has two formal parameters to allow easy configuration: `startupTime` sets the length of the startup procedure for virtual machines and `accountingPeriod` sets the length of each accounting period. In addition, the class has four fields: `accumulatedCost` stores the cost incurred by the client application up to present time, the set `billableMachines` contains the machines to be billed in the current time interval, and the sets `availableMachines` and `runningMachines` contain the created but not currently running and the running machines, respectively. The empty set is denoted `EmptySet`. Let s be a set over elements of type T and let $e : T$. The following functions are defined in the functional part of Real-Time ABS: `insertElement`(s, e) returns $\{e\} \cup s$, `remove`(s, e) returns $s \setminus \{e\}$, and `take`(s) returns some e such that $e \in s$.

The methods for resource management move machines between these sets. Any machine which is either created or running within an accounting period, is billable in that period; i.e., a machine may be both acquired and released in a period, so there may be more billable than running machines. The method `createMachine` creates a new deployment component of the given capacity and adds it to `availableMachines`. The method `acquireMachine` moves a machine from `availableMachines` to `runningMachines`. Since the machine becomes billable, it is placed in `billableMachines`. The method suspends for the duration of the `startupTime` before it returns, so the accounting includes the startup time of the machine. The method `releaseMachine` moves

```
interface CloudProvider {
  DC   createMachine(Int capacity);
  Unit acquireMachine(DC machine);
  Unit releaseMachine(DC machine);
  Int  getAccumulatedCost();
}
class CloudProvider (Int startupTime, Int accountingPeriod)
    implements CloudProvider {
  Int accumulatedCost = 0; Set<DC> billableMachines = EmptySet;
  Set<DC> availableMachines = EmptySet;
  Set<DC> runningMachines = EmptySet;

  DC createMachine(Int r) {
    DC dc = new DeploymentComponent("", set[CPUCapacity(r)]);
    availableMachines = insertElement(availableMachines, dc);
    return dc;
  }
  Unit acquireMachine(DC dc) {
    billableMachines = insertElement(billableMachines, dc);
    availableMachines = remove(availableMachines, dc);
    runningMachines = insertElement(runningMachines, dc);
    await duration(startupTime, startupTime);
  }
  Unit releaseMachine(DC dc) {
    runningMachines = remove(runningMachines, dc);
    availableMachines = insertElement(availableMachines, dc);
  }
  Int getAccumulatedCost(){ return accumulatedCost; }
  Unit run() {
    while (True) {
      await duration(accountingPeriod, accountingPeriod);
      Set<DeploymentComponent> billables = billableMachines;
      while (~(billables == EmptySet)) {
        DeploymentComponent dc = take(billables);
        billables = remove(billables,dc); Int capacity = dc.total("CPU");
        accumulatedCost = accumulatedCost+(accountingPeriod*capacity);
      }
      billableMachines = runningMachines;
}}}
```

Fig. 2. The CloudProvider class in Real-Time ABS

a machine from `runningMachines` to `availableMachines`. The machine remains billable for the current accounting period.

The `run` method of the cloud provider implements the accounting of incurred resource usage for the client application. The method suspends for the duration of the accounting period, after which all machines in `billableMachines` are billed by adding their resource capacity for the duration of the accounting period to `accumulatedCost`. Remark that Real-Time ABS has a run-to-completion semantics which guarantees that the loop in `run` will be executed after every accounting period. After accounting is finished, only the currently running machines are already billable for the next period. These are copied into `billableMachines` and the `run` method suspends for the next accounting period.

Module	Description
mImgtbl	Extract geometry information from a set of FITS headers and create a metadata table from it.
mOverlaps	Analyze an image metadata table to determine which images overlap on the sky.
mProject	Reproject a FITS image.
mProjExec	Reproject a set of images, running *mProject* for each image.
mDiff	Perform a simple image difference between a pair of overlapping images.
mDiffExec	Run *mDiff* on all the overlap pairs identified by *mOverlaps*.
mFitplane	Fit a plane (excluding outlier pixels) to an image. Used on the difference images generated by *mDiff*.
mFitExec	Run *mFitplane* on all overlapping pairs. Creates a table of image-to-image difference parameters.
mBgModel	Modeling/fitting program which uses the image-to-image difference parameter table to interactively determine a set of corrections to apply to each image to achieve a "best" global fit.
mBackground	Remove a background from a single image
mBgExec	Run *mBackground* on all the images in the metadata table.
mAdd	Co-add the reprojected images to produce an output mosaic.

Fig. 3. The modules of the Montage case study

4 Case Study: The Montage Toolkit

Montage is a portable software toolkit for generating science-grade mosaics by composing multiple astronomical images [24]. Montage is modular and can be run on a researcher's desktop machine, in a grid, or on a cloud. Due to the high volume of data in a typical astronomical dataset and the high resolution of the resulting mosaic, as well as the highly parallelizable nature of the needed computations, Montage is a good candidate for cloud deployment. In [19], Deelman et al. present simulations of cloud deployments of Montage and the cost of creating mosaics with different deployment scenarios, using the specialized simulation tool GridSim [9].

This section describes the architecture of the Montage system and how it was modeled in Real-Time ABS. We explain how costs were associated to the different parts of the model. The results obtained by simulations of the model in the Real-Time ABS interpreter are compared to those obtained in the specialized simulator. Finally, more fine-grained dynamic resource management, not considered in the previous work [19], is proposed and compared to previous scenarios.

4.1 The Problem Description

Creating a mosaic from a set of input images involves a number of tasks: first reprojecting the images to a common projection, coordinating system and scale, then rectifying the background radiation in all images to a common flux scale

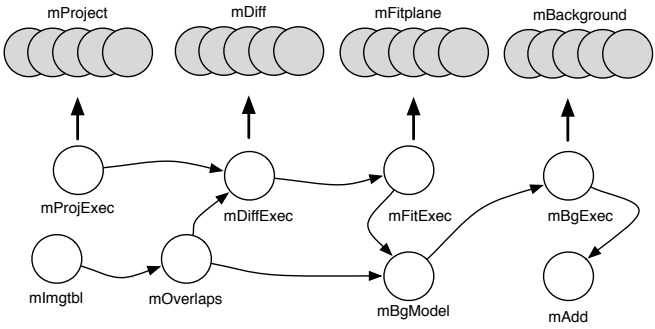


Fig. 4. Montage abstract workflow

and background level, and finally co-adding the reprojected background-rectified images into a final mosaic. The tasks exchange data in the format FITS, which encapsulates image data and meta-data. These tasks are implemented by a number of Montage modules [24], which are listed and described in Fig. 3. These modules can be run individually or combined in a workflow, locally or remotely on a grid or a cloud. Fig. 4 depicts the dataflow dependencies between the modules in a typical Montage workflow [19]. These dependencies show which jobs can be parallelized on multiprocessor systems, grids, or cloud services.

Simulation results for running Montage on the *Amazon* cloud with the workflow depicted in Fig. 4 have been published in [19], including cost measurements for CPU and storage resources. The simulation tool GridSim [9] was used to study the trade-offs between cost and performance for different execution and resource provisioning scenarios when running Montage in a cloud service provider.

We model and analyze the same abstract workflow architecture of Montage based on the model of cloud provisioning presented in Section 3, as a means to validate the presented formal model of cloud provisioning in Real-Time ABS. In particular, we consider the case in which Montage processes multiple input images in parallel. Our model abstracts from the implementation details of the manipulation of images, replacing them with abstract statements and cost annotations. One important result of [19] is that computation cost dominates storage and data transfer cost for the Montage workload by 2-3 orders of magnitude, which allows us to focus on CPU usage alone.

4.2 A Model of the Montage Workflow in Real-Time ABS

The Core Modules. The Montage core modules that execute atomic tasks (i.e., `mProject`, `mDiff`, `mFitplane`, `mBgModel`, `mBackground`, `mAdd`, `mImgtbl`, and `mOverlaps`) are modeled as methods inside a class `CalcServer` which implements the `CalcServer` interface shown in Fig. 5. In the methods of this class, cost annotations are used to specify the costs of executing atomic tasks. The images considered in the case study have a constant size, so it is sufficient to use a constant cost for the atomic tasks. Lacking precise cost estimates for

```
interface CalcServer {
  DeploymentComponent getDC();
  MetadataT mImgtbl(List<FITS> i);
  MetadataT mOverlaps(MetadataT mt);
  FITS mProject(FITS image);
  FITSdf mDiff (FITS image1, FITS image2);
  FITSfit mFitplane (FITSdf df);
  CorrectionT mBgModel(Image2ImageT diffs, MetadataT ovlaps);
  FITS mBackground (Int correction,FITS image );
  FITS mAdd (List<FITS> images); }

class CalcServer implements CalcServer {
  ...
  FITS mBackground (Int correction,FITS image ){
    [Cost: 1] FITS result = correctFITS(image,correction);
    return result;
  }
... }
```

Fig. 5. `CalcServer` interface and class in Real-Time ABS

the individual tasks, we consider an abstract cost model in which each atomic task is assigned the cost of 1 resource. (This cost model could be further refined; although some timing measurements are given in [24], these are not detailed enough for this purpose.) The code for one such atomic task inside the `CalcServer` class is shown in Fig. 5.

Resource Management. The workflow process does not interact with the different instances of `CalcServer` directly. Instead, tasks are sent to an instance of `ApplicationServer` which acts a broker for the preallocated machine instances and distributes tasks to free machines. The `ApplicationServer` interface, partly shown in Fig. 6, provides the workflow with means to start the parallelizable tasks (i.e., `mProjExec`, `mDiffExec`, `mFitExec` and `mBgExec`) and distributes the atomic tasks (e.g., `mDiff`) to instances of `CalcServer`. Atomic tasks are sent directly to one calculation server. Two fields `activeMachines` and `servers` keep track of the number of active jobs on each created machine and the order in which servers get jobs, respectively. Surrounding every call to a calculation server the auxiliary methods `getServer` and `dropServer` do the bookkeeping and resource management of the virtual machines. Asynchronous method calls to the future variables `fimage` and `fnewimages`, and task suspension are used to keep the application server responsive.

Our model defines algebraic data types `FITS`, `FITSdf`, `FITSfit`, as well as the list `MetadataT` and the maps `CorrectionT` and `Image2ImageT` to represent the input and output data at the different stages of the workflow; for example, `FITS` is a data type which represents image archives in FITS format, which is constructed from an abstract representation of metadata and of image data. This data can be used to keep track of data flow and abstractions of calculation results. The empty list and map are denoted `Nil` and `EmptyMap`. On lists, the constructor `Cons`(h, t) takes as arguments an element h and a list t;

```
interface ApplicationServer {
  FITS mAdd (List<FITS> images);
  List<FITS>  mProjExec(List<FITS> images);
  List<FITSdf> mDiffExec (MetadataT metatable, List<FITS> images);
  Image2ImageT mFitExec(List<FITSdf> dfs);
  List<FITS> mBgExec (CorrectionT corrections, List<FITS> images);
  ... }

class ApplicationServer(CloudProvider provider)
    implements ApplicationServer {
  List<CalcServer> servers = Nil; Map<DC,Int> activeMachines = EmptyMap;
  ...
  List<FITS> mBgExec(CorrectionT corrections,List<FITS> images) {
    List<FITS> newimages = Nil;
    if (isEmpty(images)==False) {
      FITS image = head(images);
      Int correction = lookupDefault(corrections,getId(image), 0);
      CalcServer b = this.getServer();
      Fut<FITS> fimage = b!mBackground (correction,image);
      Fut<List<FITS>> fnewimages=this!mBgExec(corrections,tail(images));
      await fimage?; FITS tmpimage = fimage.get;
      this.dropServer(b);
      await fnewimages?; List<FITS> newtmpimages = fnewimages.get;
      newimages = Cons(tmpimage, newtmpimages);}
    return newimages;}
... }
```

Fig. 6. The `ApplicationServer` interface and class (abridged)

`head(Cons`$(h,t)) = h$ and `tail(Cons`$(h,t)) = t$. The function `isEmpty`(l) returns true if l is the empty list. On maps, the function `lookupDefault`(m,k,v) returns the value bound to k in m if the key k is bound in m, and otherwise it returns the default value v.

4.3 Simulation Results

We simulated a workload equivalent to the *Montage 1* scenario described in [19]. As in that paper, the simulations were run on deployment scenarios ranging from 1 to 128 virtual machine instances, where all the machines were started up prior to the simulations (i.e., the `startupTime` parameter of the `CloudProvider` class in our model has value 0). Both simulation approaches exhibit the expected geometric downward progression of execution time when going from 1 to 128 machines, and roughly half an order of magnitude increase in cost. In our first simulation runs, the execution cost (measured in simulated machine-minutes) increased a little over two-fold over the full simulation range, versus closer to a six-fold increase ("60 cents [...] versus almost 4$") in [19]. To explain this difference, we theorized that the observed lower cost may have resulted from better machine allocation strategies in our model—the virtual machines were eagerly released by the `ApplicationServer` class when no more work was available to them, instead of being kept running until all computations finished.

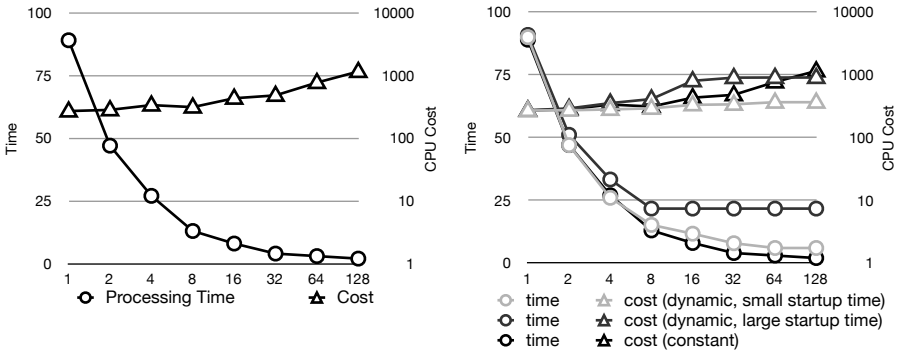


Fig. 7. Execution costs and times of simulation. The *Montage 1* scenario (left figure) is compared to dynamic resource management (right figure). The costs are presented on a logarithmic scale for easier comparison with the results of [19].

To test this hypothesis, the `ApplicationServer` class was modified to keep all instances running during the whole computation task. Using this allocation strategy, we observed a cost increase of 4.27 from 1 to 128 computation servers, which is more in line with the results obtained using GridSim. Fig. 7 (left) shows the simulation results of the modified model. The authors of [19] later confirmed in private communication that our hypothesis about the setup of the GridSim simulation scenario was indeed correct.

In order to further investigate the initial results involving dynamic startup and shutdown of machine instances, we refine our model by introducing startup times for virtual machines. Fig. 7 (right) compares the previous static deployment scenario (constant) with two dynamic resource management scenarios with varying startup times for virtual machines. One scenario models machine startup times of roughly one tenth of the time needed for performing a basic task, the other startup times roughly as large as basic task times. It can be seen that the cost of running a single job in the Montage system can be substantially reduced by switching off unused machines, given that the cost of starting machines is dominated by the actual calculations taking place, with almost no loss in time. On the other hand, if starting a machine is significantly slower than executing a basic task, it can be seen that both cost and time of the dynamic scenario are worse than when initially starting all machines in the static scenario of the considered workflow except in the case of severe over-provisioning of machines.

5 Related Work

The concurrency model of ABS is based on concurrent objects and Actor-based computation, in which software units with encapsulated processors communicate asynchronously (e.g., [1,5,14,23,26]). Their inherent compositionality allows concurrent objects to be naturally distributed on different locations, because only

the local state of a concurrent object is needed to execute its methods. In previous work, the authors have introduced *deployment components* as a formal modeling concept to capture restricted resources shared between concurrent object groups and shown how components with parametric resources naturally model different deployment architectures [28], extended the approach with resource reallocation [27], and combined it with static cost analysis [4]. This paper complements our previous work by using deployment components to model cloud-based scenarios and the development of the Montage case study. A companion paper [18] further applies the approach of this paper to an industrial case study.

Techniques for prediction or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [20]. A survey of model-based performance analysis techniques is given in [7]. Formal systems using process algebra, Petri Nets, game theory, and timed automata have been used in the embedded software domain (e.g., [15,21]). Real-Time ABS combines *explicit* time modeling with duration statements with *implicit* measurements of time already at the modeling level, which is made possible by the combination of costs in the application model and capacities in the deployment components.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance, and time, Petriu and Woodside [32] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects an operation's set of resources. CSM aims to bridge the gap between UML and techniques to generate performance models [7]. Closer to our work is M. Verhoef's extension of VDM++ for embedded real-time systems [33], in which static architectures are explicitly modeled using CPUs and buses. The approach uses fixed resources targeting the embedded domain, namely processor cycles bound to the CPUs, while we consider more general resources for arbitrary software. Verhoef's approach is also based on abstract executable modeling, but the underlying object models and operational semantics differ. VDM++ has multi-thread concurrency, preemptive scheduling, and a strict separation of synchronous method calls and asynchronous signals, in contrast to our work with concurrent objects, cooperative scheduling, and caller-decided synchronization.

Related work on simulation tools for cloud computing are typically reminiscent of network simulators. A number of testing techniques and tools for cloud-based software systems are surveyed in [6]. In particular, CloudSim [13] and ICanCloud [31] are simulation tools using virtual machines to simulate cloud environments. CloudSim is a fairly mature tool which has already been used for a number of papers, but it is restricted to simulations on a single computer. In contrast, ICanCloud supports distribution on a cluster. Additionally CloudSim was originally based on GridSim [9], a toolkit for modeling and simulations of heterogeneous Grid resources. EMUSIM [12] is an integrated tool that uses AEF [11] (Automated Emulation Framework) to estimate performance and costs for an

application by means of emulations to produce improved input parameters for simulations in CloudSim. Compared to these approaches, our work is based on a formal semantics and aims to support the developer of software applications for cloud-based environments at an early phase in the development process.

Another interesting line of research is static cost analysis for object-oriented programs (e.g., [3,22]) Most tools for cost analysis only consider sequential programs, and assume that the program is fully developed before cost analysis can be applied. COSTABS [2] is a cost analysis tool for ABS which supports concurrent object-oriented programs. Our approach, in which the modeler specifies cost in cost annotations, could be supported by COSTABS to automatically derive cost annotations for the parts of a model that are fully implemented. In collaboration with Albert *et al.*, we have applied this approach for memory analysis of ABS models [4]. However, the full integration of COSTABS in our tool chain and the software development process remain future work.

6 Conclusion

This paper develops a model in Real-Time ABS of a cloud provider which offers virtual machines with given CPU capacities to a client application. Virtual machines are modeled as deployment components with given CPU capacities, and the cloud provider offers methods for resource management of virtual machines to client applications. The proposed model has been validated by means of a case study of the Montage toolkit, in which a typical Montage workflow was formalized. This formalization allows different user scenarios and deployment models to easily expressed and compared by means of simulations using the Real-Time ABS interpreter. The results from these simulations were comparable to those obtained for the Montage case study using specialized simulators, which suggests that models using abstract behavioral specification languages such as Real-Time ABS can be used to estimate cloud deployment costs for realistic systems.

Real-Time ABS aims to support the developer of client applications for cloud-based deployment, and in particular to facilitate the development of strategies for virtualized resource management at early stages in the development process. We are not aware of similar work addressing the formal modeling of virtualized resource management and cloud computing from the client application perspective. With the increasing focus on cloud-based deployment of general purpose software, such support could become very useful for software developers.

This paper focused on the formalization of cloud provisioning and simulations of the executable model. The presented work can be extended in a number of directions. In particular, we are interested in how to combine different virtualized resources in the same model to estimate combined costs of, e.g., computations, storage, bandwidth, and power consumption. Another extension is to strengthen the tool-based analysis support for Real-Time ABS. An integration with cost analysis tools such as COSTABS would assist the developer in providing cost annotations in the model. Furthermore, we plan to investigate symbolic execution techniques for Real-Time ABS, which would allow stronger automated analysis results than those considered here. Finally, an integration of

QoS contracts with the interfaces of Real-Time ABS could form a basis for analysis abstract behavioral specifications with respect to service-level agreements.

Acknowledgment. We thank G. Bruce Berriman and Ewa Deelman for helping us with additional details of the Montage case study.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge (1986)
2. Albert, E., Arenas, P., Genaim, S., Gómez-Zamalloa, M., Puebla, G.: COSTABS: a cost and termination analyzer for ABS. In: Proc. Workshop on Partial Evaluation and Program Manipulation (PEPM 2012), pp. 151–154. ACM (2012)
3. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
4. Albert, E., Genaim, S., Gómez-Zamalloa, M., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: Simulating Concurrent Behaviors with Worst-Case Cost Bounds. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 353–368. Springer, Heidelberg (2011)
5. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
6. Bai, X., Li, M., Chen, B., Tsai, W.-T., Gao, J.: Cloud testing tools. In: Proc. 6th Symposium on Service Oriented System Engineering, pp. 1–12. IEEE (2011)
7. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Transactions on Software Engineering 30(5), 295–310 (2004)
8. Bjørk, J., de Boer, F.S., Johnsen, E.B., Schlatte, R., Tapia Tarifa, S.L.: User-defined schedulers for real-time concurrent objects. Innovations in Systems and Software Engineering (2012), http://dx.doi.org/10.1007/s11334-012-0184-5
9. Buyya, R., Murshed, M.: GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. Concurrency and Computation: Practice and Experience 14, 1175–1220 (2002)
10. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599–616 (2009)
11. Calheiros, R.N., Buyya, R., De Rose, C.A.F.: Building an automated and self-configurable emulation testbed for grid applications. Software: Practice and Experience 40(5), 405–429 (2010)
12. Calheiros, R.N., Netto, M.A., De Rose, C.A.F., Buyya, R.: EMUSIM: an integrated emulation and simulation environment for modeling, evaluation, and validation of performance of cloud computing applications. Software: Practice and Experience (2012)
13. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Software, Practice and Experience 41(1), 23–50 (2011)
14. Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer (2005)
15. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource Interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)

16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.): All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
17. de Boer, F.S., Clarke, D., Johnsen, E.B.: A Complete Guide to the Future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
18. de Boer, F.S., Hähnle, R., Johnsen, E.B., Schlatte, R., Wong, P.Y.H.: Formal Modeling of Resource Management for Cloud Architectures: An Industrial Case Study. In: De Paoli, F., Pimentel, E., Zavattaro, G. (eds.) ESOCC 2012. LNCS, vol. 7592, pp. 91–106. Springer, Heidelberg (2012)
19. Deelman, E., Singh, G., Livny, M., Berriman, G.B., Good, J.: The cost of doing science on the cloud: The Montage example. In: Proc. High Performance Computing (SC 2008), pp. 1–12. IEEE/ACM (2008)
20. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. ICSE 2009, pp. 111–121. IEEE (2009)
21. Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Information and Computation 205(8), 1149–1172 (2007)
22. Gulwani, S., Mehra, K.K., Chilimbi, T.M.: Speed: Precise and Efficient Static Estimation of Program Computational Complexity. In: Proc. POPL 2009, pp. 127–139. ACM (2009)
23. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
24. Jacob, J.C., Katz, D.S., Berriman, G.B., Good, J., Laity, A.C., Deelman, E., Kesselman, C., Singh, G., Su, M.-H., Prince, T.A., Williams, R.: Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking. Intl. Journal of Computational Science and Engineering 4(2), 73–87 (2009)
25. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 142–164. Springer, Heidelberg (2011)
26. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
27. Johnsen, E.B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Dynamic Resource Reallocation between Deployment Components. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 646–661. Springer, Heidelberg (2010)
28. Johnsen, E. B., Owe, O., Schlatte, R., Tapia Tarifa, S.L.: Validating Timed Models of Deployment Components with Parametric Concurrency. In: Beckert, B., Marché, C. (eds.) FoVeOOS 2010. LNCS, vol. 6528, pp. 46–60. Springer, Heidelberg (2011)
29. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. Intl. Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
30. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
31. Nuñez, A., Vázquez-Poletti, J., Caminero, A., Castañé, G., Carretero, J., Llorente, I.: iCanCloud: A flexible and scalable cloud infrastructure simulator. Journal of Grid Computing 10, 185–209 (2012)
32. Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and System Modeling 6(2), 163–184 (2007)
33. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and Validating Distributed Embedded Real-Time Systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)

Specification and Model Checking of the Chandy and Lamport Distributed Snapshot Algorithm in Rewriting Logic

Kazuhiro Ogata and Phan Thi Thanh Huyen

School of Information Science, JAIST
{ogata,huyenttp}@jaist.ac.jp

Abstract. Many model checkers have been developed and then many case studies have been conducted by applying them to mechanical analysis of systems including distributed systems, protocols and algorithms. To the best of our knowledge, however, there are few case studies in which the Chandy & Lamport distributed snapshot algorithm is mechanically analyzed with model checkers. We think that this is because it is not straightforward to express the significant property that the algorithm should enjoy in LTL and CTL. In this paper, we describe how to specify the algorithm in Maude, a specification and programming language based on rewriting logic, and how to model check the significant property with the Maude search command, which demonstrates the power of the command. The case study also demonstrates the importance of case analysis in specification.

Keywords: distributed snapshot, distributed system, Maude, model checking, the search command.

1 Introduction

Many model checkers such as symbolic model checkers, explicit-state model checkers and SAT/SMT-based bounded model checkers have been proposed[1,2,3]. Accordingly, many case studies have been conducted by applying them to mechanical analysis of systems including distributed systems, protocols and algorithms[4,5,6]. To the best of our knowledge, however, there are few case studies in which the Chandy & Lamport distributed snapshot algorithm[7] is mechanically analyzed with model checkers. We think that this is because it is not straightforward to express the most significant property that the algorithm should enjoy in standard temporal logics such as LTL and CTL.

Let s_1, s_* and s_2 be the state (called the start state) when a distributed snapshot starts being taken, the snapshot, and the state (called the finish state) when the snapshot completes being taken, respectively. The most significant property is that s_* is always reachable from s_1 and s_2 is always reachable from s_*. To check the property, all needed to do is to check if s_* is reachable from s_1 and s_2 is reachable from s_* whenever s_2 is obtained, namely that the algorithm terminates. If computations, namely sequences of states, are first-class objects,

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 87–102, 2012.

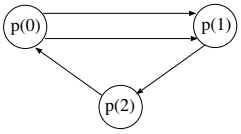


Fig. 1. The 3-process & 4-channel system

the property can be expressed as an invariant property[8]. But, computations are not first-class objects in LTL and CTL. There might be some way to model the algorithm such that the property can be expressed in LTL and/or CTL. But, such a model must be too complicated to be model checked.

Maude[9] is a specification and programming language based on rewriting logic. The Maude search command can be used as a (bounded) model checker for invariant properties. Given a state s, a state pattern p and an optional condition c, the search command searches the reachable state space from s in a breadth-first manner for all states that match p such that c holds. Reachability can also be checked in the optional condition c. This makes it possible to check the property. In this paper, we describe how to specify the algorithm in Maude and how to model check the property with the search command, which demonstrates the power of the command, namely that more general invariant properties can be checked by the command than standard LTL and CTL model checkers. The case study also demonstrates the importance of case analysis in specification, which often needs to be conducted for interactive theorem proving. It is also worth noting that the formal specification of the algorithm in Maude depends on neither the number of processes nor the number of channels in the algorithm, although we need to fix them to conduct model checking.

The rest of the paper is organized as follows. §2 describes the Chandy & Lamport algorithm. §3 describes Maude. §4 describes how to specify the algorithm in Maude. §5 describes how to model check the property with the search command. §6 mentions some related work. §7 concludes the paper.

2 The Chandy and Lamport Algorithm

A distributed system (called an underlying system) whose snapshot is taken is first described. An underlying system consists of one or more processes (typically two or more processes) and directed channels that are unbounded queues from processes to processes. Given two processes, there may be two or more channels from one to the other. Fig.1 shows a system that consists of three processes and four channels, in which there are two channels from process p(0) to process p(1). Each process has its own state, and so does each channel. The state of a channel is a sequence of messages that have been sent by its source process but that have not yet been received by its destination process. Each channel is reliable in that no messages are lost, duplicated or swapped. The state of the system is the collection of the states of the processes and channels. What each process does is

- to put a message into one of its outgoing channels and/or change its state,
- to get a message from one of its non-empty incoming channels and/or change its state, or
- to change its state with neither sending nor receiving any messages.

When a snapshot is taken, each process records its own state and the state of each of its incoming channels. But, the problem is when each process should do so. If each process does so at any time when it wants, the global snapshot may be inconsistent, namely that the snapshot may not be reachable from the state when the snapshot has started being taken. Chandy and Lamport designed an algorithm that instructs each process when it does so by sending special messages called markers[7].

What each process does with respect to (w.r.t.) the algorithm is as follows:

1. The process may record its state when it has not yet received any markers. If that is the case, the process puts a marker into each of its outgoing channels before it puts any non-marker messages into the channel.
2. The process may get a marker from one of its incoming channels.
   (a) If the process has not yet recorded its state, it records its state and the state of the incoming channel as empty, and initializes the records of the other incoming channels as empty. The process then puts a marker into each of its outgoing channels before it puts any non-marker messages into the channel.
   (b) If the process has already recorded its state, it has completed the record of the incoming channel.
3. When the process has already recorded its state and gets a non-marker message from one of its incoming channel from which a marker has not yet been received, it updates the record of the incoming channel by putting the non-marker message into the record.

When each process has completed the records of its state and all of its incoming channels, a global snapshot has been taken, which is the collection of all those records.

Let s_1, s_* and s_2 be the state (called the start state) when a distributed snapshot starts being taken, the snapshot, and the state (called the finish state) when the snapshot completes being taken, respectively. The most significant property consists of two reachability properties that are called the reachability property 1 (RP1) and the reachability property 2 (RP2), respectively, in this paper. RP1 is that s_* is always reachable from s_1, and RP2 is that s_2 is always reachable from s_*.

3 Maude

Maude[9] is a specification and programming language based on rewriting logic that includes as a sub-logic membership equational logic (an extension of order-sorted equational logic). State machines (or transition systems) are specified in

rewriting logic, and their specifications are called system specifications. Data used in state machines are specified in membership equational logic. States of state machines are expressed as data such as tuples and associative-commutative collections (called soups), and state transitions are described in rewriting rules.

Basic units of Maude specifications/programs are modules. Some built-in modules are provided such as BOOL and NAT for Boolean values and natural numbers. The Boolean values are denoted as true and false, and natural numbers as $0, 1, 2, \ldots$ as usual. The corresponding sorts are Bool and Nat. Precisely, there are three sorts for natural numbers Zero, NzNat, and Nat that are for zero, non-zero natural numbers, and natural numbers that may be zero or non-zero. Sort Nat is the super-sort of Zero and NzNat.

Let us consider a simple system as an example. The system consists of two processes p and q and one channel (which is an unbounded queue) from p to q. Each process has a set of natural numbers, which is regarded as the state of the process. Initially, p's set is $\{0, 1, 2\}$, q's set is empty, and the channel is empty. p arbitrarily chooses and deletes one natural number x from its set, and puts x into the channel, which is referred to as p's action. If the channel is not empty and q's set does not contain 0, q gets the top y from the channel and adds y to its set, which is referred to as q's action. Let us specify this system, precisely a state machine modeling this system, in Maude.

A set of natural numbers is expressed as a soup of natural numbers. The corresponding sort is NSoup that is declared as a super-sort of Nat, which means that a natural number itself is also the singleton. The empty set is denoted as noNat, and the soup of n natural numbers $x_1, \ldots, x_n$ as $x_1 \ \ldots \ x_n$ whose constructor is called the juxtaposition operator or the empty syntax. noNat is declared as an identity of the constructor.

The empty channel is denoted as empChan, and the non-empty channel that consists of n natural numbers $x_1, \ldots, x_n$ as $x_1 \,|\, \ldots \,|\, x_n \,|\, \text{empChan}$. The corresponding sort is Chan.

States of p, q, and the channel are expressed as p : ns, q : ms, and c : q, respectively, where ns and ms are soups of natural numbers and q is a channel (queue) of natural numbers. p : ns, q : ms, and c : q are called observable components, and the corresponding sort is OCom. A state of the system is expressed as a soup (called a configuration) of those observable components, which is expressed as (p : ns) (q : ms) (c : q). The corresponding sort is Config that is a super-sort of OCom. The initial configuration is expressed as (p : (0 1 2)) (q : noNat) (c : empChan). Let ic be the initial configuration.

Let N be a Maude variable of sort Nat, NS be a Maude variable of sort NSoup, C be a Maude variable of sort Chan in the rest of this section.

p's action is described in the following rewriting rule:

rl [snd] : (p : (N NS)) (c : C) $\Rightarrow$ (p : NS) (c : put(C, N)) .

where snd is the label of the rewriting rule, and put takes a channel C and a natural number N and returns the channel obtained by putting N into C at the end. If a given term contains an instance of (p : (N NS)) (c : C), the instance is replaced with the corresponding instance of (p : NS) (c : put(C, N)).

q's action is described in the following rewriting rule:

crl [rec] : (c : $(N \mid C)$ (q : NS) $\Rightarrow$ (c : C) (q : $(N\ NS)$) **if** $0 \notin NS$.

This rewriting rule is conditional. The condition is $0 \notin NS$. The rule can be applied if the condition holds.

The Maude system is equipped with model checking facilities: the search command and the LTL model checker. In this paper, the search command is used. Given a state s, a state pattern p and an optional condition c, the search command searches the reachable state space from s in a breadth-first manner for all states that match p such that c holds. Such states are called solutions. The syntax is as follows:

search in M : $s \Rightarrow^* p$ **such that** c .

where M is a module in which the specification of the state machine concerned is described or available. A rewrite expression $t \Rightarrow t'$ can be used in the optional condition c. This checks if t' is reachable from t by zero or more rewrite steps with rewriting rules. This is the essence of model checking RP1 and RP2 for the Chandy & Lamport distributed snapshot algorithm.

The following search finds all states (configurations) such that they are reachable from ic and the q's set contains only 2:

search in EXPERIMENT : ic $\Rightarrow^*$ (q : 2) *CONFIG* .

where EXPERIMENT is the module in which the specification of the system we have been discussing is available. The search finds 5 solutions.

The following search finds all states (configurations) such that they are reachable from ic, the q's set contains only 2, and (p : noNat) (c : empChan) (q : (0 1 2)) is reachable from them:

search in EXPERIMENT : ic $\Rightarrow^*$ (q : 2) *CONFIG*
such that (q : 2) *CONFIG* $\Rightarrow$ (p : noNat) (c : empChan) (q : (0 1 2)) .

The search finds 3 solutions.

Note that although the reachable state space from ic is bounded, the whole state space is unbounded. The search command can be given as options the maximum number of solutions and the maximum depth of search. If the maximum number n of solutions is given, the search terminates when it finds n solutions. Therefore, even if the reachable state space from a given state is unbounded, the search command can be used and may terminate. If the maximum depth d of search is given, only the bounded reachable state space from a given state up to depth d is searched. Hence, the search command can be used as a bounded model checker. These options are not used in this paper.

4 System Specification of the Algorithm

We describe our way of modeling (formalizing) the algorithm. What is modeled (formalized) is actually distributed systems on which the algorithm is superimposed. An underlying distributed system consists of one or more processes that

are connected with directed channels that are unbounded queues. To cover all possible situations, a system may consists of one process only, and some processes have no outgoing channels, no incoming channels, or neither of them, although such a system may not be regarded as a distributed system, or may be regarded as multiple distributed systems. Processes exchange non-marker messages that are called tokens and may consume them. We suppose that the state of each process only depends on the set of tokens owned by the process. We also suppose that at most one distributed snapshot is taken in each computation of a distributed system, and there is no self-channel, namely a channel from a process to the same process.

4.1 Basic Data Used

Processes (or process identifiers) are denoted as $\mathrm{p}(0), \mathrm{p}(1), \ldots$, and the sort is Pid.

Tokens are denoted as $\mathrm{t}(0), \mathrm{t}(1), \ldots$, and the sort is Token. A marker is denoted as marker, and the sort is Marker. Sort Msg is declared as a super-sort of Token and Marker.

Sorts EmpChan, NeChan, and Chan are for the empty channel, non-empty channels, and channels that may be empty or non-empty. The empty channel is denoted as empChan. A non-empty channel that consists of n messages $m_0, m_1, \ldots, m_{n-1}$ in this order is denoted as $m_0 \,|\, m_1 \,|\, \ldots \,|\, m_{n-1} \,|\, \mathrm{empChan}$. A function put takes a channel c and a message m (namely a token or a marker), and returns the channel obtained by putting m into c at the end.

Since the state of a process only depends on the set of tokens owned by the process, the state can be expressed as the soup of tokens. The sort for soups of tokens is PState that is also declared as a super-sort of sort Token. The empty soup is denoted as noToken. The soup of n tokens $\mathrm{t}(0), \mathrm{t}(1), \ldots, \mathrm{t}(n-1)$ is denoted as $\mathrm{t}(0)\ \mathrm{t}(1)\ \ldots\ \mathrm{t}(n-1)$. Note that noToken is declared as an identity of the constructor (the juxtaposition operator) of soups of tokens.

Each process has not yet started the algorithm, has started but not completed it, or completed it. Those situations are denoted as notYet, started and completed, respectively. The corresponding sort is Prog.

4.2 Observable Components and (Meta) Configurations

The state of an underlying system consists of the state of each process and the state of each channel. The state ps of a process p is denoted as $\text{p-state}[p] : ps$, and the state cs of a channel from a process p to a process q is denoted as $\text{c-state}[p, q, n] : cs$, where n is a natural number. Since there may be more than one channel from p to q, n is used to identify one of them. "$\text{p-state}[p] : ps$" and "$\text{c-state}[p, q, n] : cs$" are called observable components. The corresponding sort is OCom. The state of a system is expressed as a soup of observable components that is called a configuration. The corresponding sort is Config that is a super-sort of OCom. The empty configuration is denoted as empConfig that is an identity of the constructor of soups of observable components.

Example 1 (A configuration of the 3-process & 4-channel system). Let us consider the 3-process & 4-channel system in Fig. 1. We suppose that p(0) has one token t(1), the other processes have no token, one channel from p(0) to p(1) consists of one token t(0), and the other channels are empty. The state is expressed as the configuration:

(p-state[p(0)] : t(1)) (p-state[p(1)] : noToken) (p-state[p(2)] : noToken)
(c-state[p(0), p(1), 0] : (t(0) | empChan)) (c-state[p(0), p(1), 1] : empChan)
(c-state[p(1), p(2), 0] : empChan) (c-state[p(2), p(0), 0] : empChan)

We need to have more information to take into account the algorithm superimposing an underlying system. We need to record the start state, the snapshot, and the finish state. Moreover, we also need to have some control information for the algorithm.

The states of an underlying distributed system, the start state, the finish state, and the snapshot are expressed as base-state(...), start-state(...), finish-state(...), and snapshot(...), respectively, where ... is a soup of p-state and c-state observable components. Those are called meta configuration components and the sort is MCComp.

In addition to p-state and c-state observable components, we use the following observable components of control information:

- "cnt : n" – n is the number of processes that have not yet completed the algorithm. When n becomes 0, a distributed snapshot has been taken.
- "prog[p] : pg" – It indicates that a process p has not yet started, has started, or completed the algorithm. pg is notYet, started, or completed.
- "#ms[p] : n" – n is the number of incoming channels to a process p from which markers have not yet been received. When n becomes 0, p has received markers from all the incoming channels, implying that p completes the algorithm if p has one or more incoming channels. Note that p may have no incoming channels and then n may be 0 even in initial states.
- "done[p, q, n] : b" – b is either true or false. If b is true, a process q has received a marker from the incoming channel identified by n from a process p to q. Otherwise, q has not.
- "consume : b" – b is either true or false. If b is true, tokens may be consumed. Otherwise, tokens are not.

The control information is expressed as control(...) that is also a meta configuration component, where ... is a soup of cnt, prog, #ms, done and consume observable components. When the content in each prog component is completed, a snapshot has been taken. Hence, the cnt component seems redundant. But, the component can make the system specification less complicated. This is why it is used.

A state of an underlying distributed system that is superimposed by the algorithm is expressed as a soup of meta configuration components:

base-state(bc) start-state(sc) finish-state(fc) snapshot(ssc) control(ctl)

which is called a meta configuration. The corresponding sort is MConfig that is a super-sort of MCComp. Initially, all of *sc*, *fc* and *ssc* are empConfig. If *sc* is not empConfig, a distributed snapshot has started being taken. If *fc* is not empConfig, a distributed snapshot has been taken and then *ssc* is the snapshot.

Example 2 (A meta configuration of the 3-process & 4-channel system). Let us consider the 3-process & 4-channel system again. Initially, however, p(0) has the two tokens t(0) and t(1), the other processes have no tokens, and each channel is empty. Tokens may be consumed. The initial meta configuration is as follows:

base-state((p-state[p(0)] : (t(0) t(1))) (p-state[p(1)] : noToken)
(p-state[p(2)] : noToken)
(c-state[p(0), p(1), 0] : empChan) (c-state[p(0), p(1), 1] : empChan)
(c-state[p(1), p(2), 0] : empChan) (c-state[p(2), p(0), 0] : empChan))
start-state(empConfig)
finish-state(empConfig)
snapshot(empConfig)
control((cnt : 3) (#ms[p(0)] : 1) (#ms[p(1)] : 2) (#ms[p(2)] : 1)
(done[p(0), p(1), 0] : false) (done[p(0), p(1), 1] : false)
(done[p(1), p(2), 0] : false) (done[p(2), p(0), 0] : false)
(prog[p(0)] : notYet) (prog[p(1)] : notYet) (prog[p(2)] : notYet)
(consume : true))

4.3 State Transitions for Underlying Systems and the Algorithm

What each process in the underlying systems superimposed by the algorithm does is as follows:

1. The process may consume a token owned by it. Accordingly, it changes its state.
2. The process may put a token owned by it in one outgoing channel if it has some outgoing channels. Accordingly, it changes its state.
3. The process may get a token from one non-empty incoming channel if it has some non-empty incoming channels. Accordingly, it changes its state.
4. The process may start the algorithm when it has not yet received any markers. It records its state, initializes the states of its incoming channels as empty if any, and puts markers in its outgoing channels if any.
5. The process may get a marker from one incoming channel if it has some incoming channel. If it has already started the algorithm, it has completed the record of the incoming channel. Moreover, if it has received markers from all the incoming channels, it has locally completed the algorithm. If it has not yet started, it records its state and the state of the incoming channel as empty, and initializes the states of the other incoming channels as empty if any. Then, it puts markers in its outgoing channels if any. If it has only one incoming channel, it has locally completed the algorithm.

In the rest of the paper, BC, CC, SC and SSC are Maude variables of sort Config, P and Q are Maude variables of sort Pid, T is a Maude variable of sort Token, PS is a Maude variable of sort PState, N is a Maude variable of sort Nat, C and C' are Maude variables of sort Chan, and NzN and NzN' are Maude variables of sort NzNat.

Consumption of Tokens. The consumption of tokens is described by the following rewriting rule:

rl [chgStt] :
base-state((p-state[P] : (T PS)) BC)
finish-state(empConfig) control((consume : true) CC)
⇒
base-state((p-state[P] : PS) BC)
finish-state(empConfig) control((consume : true) CC) .

When a distributed snapshot has been taken, namely that the content of the finish-state meta configuration component is not empConfig, then we intentionally stop the underlying computation because we want to reduce the size of the reachable state space. This is why we have finish-state(empConfig) on both sides of the rule.

Sending of Tokens. The sending of tokens is described by the following rewriting rule:

rl [sndTkn] :
base-state((p-state[P] : (T PS)) (c-state[P, Q, N] : C) BC)
finish-state(empConfig)
⇒
base-state((p-state[P] : PS) (c-state[P, Q, N] : put(C, T)) BC)
finish-state(empConfig) .

Receipt of Tokens. When a process receives a token from an incoming channel, we need to take into account the following four cases:

(3-1) The process has not yet started the algorithm.
(3-2) The process has completed the algorithm.
(3-3) The process has started the algorithm, not yet completed it, and has not yet received a marker from the incoming channel.
(3-4) The process has started the algorithm, not yet completed it, and has already received a marker from the incoming channel.

In this paper, we show the rewriting rule for case (3-3):

rl [recTkn&started&~done] :
base-state((p-state[P] : PS) (c-state[Q, P, N] : T | C) BC)
snapshot((c-state[Q, P, N] : C') SSC)

finish-state(empConfig)
control((prog[P] : started) (done[Q, P, N] : false) CC)
⇒
base-state((p-state[P] : (T PS)) (c-state[Q, P, N] : C) BC)
snapshot((c-state[Q, P, N] : put(C', T)) SSC)
finish-state(empConfig)
control((prog[P] : started) (done[Q, P, N] : false) CC) .

When a process P starts the algorithm, it initializes the record of each incoming channel (identified by a natural number N) from each process Q unless a marker has been received from the channel. The initialization is described by adding "c-state[Q, P, N] : empChan" into the snapshot meta configuration component. For case (3-3), such a record is updated by putting the received token. For the other three cases, it is not necessary to update such a record.

Record of Process States. If a process has already received a marker, it has already recorded its state as well. Hence, we only need to take into account the case in which a process has not yet received any markers. When a process records its state in the case, the case is split into two sub-cases:

(4-1) The process globally initiates the algorithm, namely the first process that records its state in the system.
(4-2) The process does not, namely that there exists another process that has globally initiated the algorithm.

Case (4-1) is further split into three sub-cases:

(4-1-1) The underlying system only consists of the process.
(4-1-2) The system consists of more than one process, and the process does not have any incoming channels.
(4-1-3) The system consists of more than one process, and the process has one or more incoming channels.

Case (4-2) is further split into three sub-cases:

(4-2-1) The process does not have any incoming channels, and there are no processes except for the process that have not completed the algorithm.
(4-2-2) The process does not have any incoming channels, and there are some other processes that have not completed the algorithm.
(4-2-3) The process has some incoming channels.

For case (4-1-1), the algorithm will be completed. For case (4-1-2), the process will locally complete the algorithm. For case (4-1-3), we have the following rewriting rule:

rl [start&#ms>0] :
base-state((p-state[P] : PS) BC)
start-state(empConfig) snapshot(empConfig)

control((prog[P] : notYet) (#ms[P] : NzN') CC)
⇒
base-state((p-state[P] : PS) bcast($BC, P,$ marker))
start-state((p-state[P] : PS) BC)
snapshot((p-state[P] : PS) $inchans(BC, P)$)
control((prog[P] : started) (#ms[P] : NzN') CC) .

where bcast($BC, P,$ marker) puts markers in all the outgoing channels from process P, and inchans(BC, P) initializes the states of all the incoming channels. start-state(empConfig) indicates that the process P is the first that starts taking a distributed snapshot. (#ms[P] : NzN') means that the process P has one or more incoming channels and then the system consists of more than one process because NzN' is a non-zero natural number and the number of the process P's incoming channels if P has not started the algorithm, which is indicated by (prog[P] : notYet). The start state is recorded as "(p-state[P] : PS) BC" in the start-state meta configuration component. The process P's state is recorded and the state of each incoming channel to P is initialized as "(p-state[P] : PS) $inchans(BC, P)$" in the snapshot meta configuration component.

For case (4-2-1), the algorithm will be completed. For case (4-2-2), the process will locally complete the algorithm. For case (4-2-3), we have the following rewriting rule:

crl [record&#ms>0] :
base-state((p-state[P] : PS) BC)
start-state(SC) snapshot(SSC)
control((prog[P] : notYet) (#ms[P] : NzN') CC)
⇒
base-state((p-state[P] : PS) bcast($BC, P,$ marker))
start-state(SC) snapshot((p-state[P] : PS) inchans(BC, P) SSC)
control((prog[P] : started) (#ms[P] : NzN') CC)
if $SC \neq$ empConfig .

The condition $SC \neq$ empConfig indicates that the process P is not the first that starts the algorithm.

Receipt of Markers. When a process receives a marker from an incoming channel, we first need to take into account the following two cases:

(5-1) The process has not yet started the algorithm.
(5-2) The process has already started the algorithm.

Case (5-1) is further split into three sub-cases:

(5-1-1) The process has only one incoming channel, and there are no processes that have not yet completed the algorithm except for the process, which implies that the process does not have any outgoing channels.
(5-1-2) The process has only one incoming channel, and there are some other processes that have not yet completed the algorithm.

(5-1-3) The process has more than one incoming channel.

Case (5-2) is further split into three sub-cases:

(5-2-1) There are no incoming channels from which markers have not been received except for the incoming channel, and there are no processes that have not yet completed the algorithm except for the process.

(5-2-2) There are no incoming channels from which markers have not been received except for the incoming channel, and there are some other processes that have not yet completed the algorithm.

(5-2-3) There are some other incoming channels from which markers have not been received.

In this paper, we show the rewriting rules for cases (5-1-3) and (5-2-1):

```
crl [recMkr&notYet&#ms>1] :
  base-state((p-state[P] : PS) (c-state[Q,P,N] : marker | C) BC)
  snapshot(SSC)
  control((prog[P] : notYet) (#ms[P] : NzN') (cnt : NzN)
          (done[Q,P,N] : false) CC)
  =>
  base-state((p-state[P] : PS) (c-state[Q,P,N] : C) bcast(BC,P,marker))
  snapshot((p-state[P] : PS) (c-state[Q,P,N] : empChan)
           inchans(BC,P) SSC)
  control((prog[P] : started) (#ms[P] : sd(NzN',1)) (cnt : NzN)
          (done[Q,P,N] : true) CC)
if NzN' > 1 .
```

where sd, which stands for symmetric difference, takes two natural numbers x, y, and returns $x - y$ if $x > y$ and $y - x$ otherwise. Note that when a process receives a marker from an incoming channel, the natural number in the #ms observable component must be greater than zero and the natural number in the cnt observable component must be greater than zero. This is why we have (cnt : NzN) on both sides.

```
rl [recMkr&started&#ms=1&cnt=1] :
  base-state((p-state[P] : PS) (c-state[Q,P,N] : marker | C) BC)
  finish-state(empConfig)
  control((prog[P] : started) (#ms[P] : 1) (cnt : 1) (done[Q,P,N] : false) CC)
  =>
  base-state((p-state[P] : PS) (c-state[Q,P,N] : C) BC)
  finish-state((p-state[P] : PS) (c-state[Q,P,N] : C) BC)
  control((prog[P] : completed) (#ms[P] : 0) (cnt : 0)
          (done[Q,P,N] : true) CC) .
```

When the process P receives the marker, a global snapshot has been taken because of the assumption (see **(5-2-1)**). The receipt of the marker by P does not affect (the contents of) the global snapshot. This is why the rewriting rule

does not have any snapshot meta configuration components. The finish state is recorded as (p-state[P] : PS) (c-state[Q, P, N] : C) BC in the finish-state meta configuration component.

Note that the system specification depends on neither the number of processes nor the number of channels. To model check RP1 and RP2, however, we need to fix those numbers, which are described in initial meta configurations.

A meta configuration is a global view of the system (a distributed system superimposed by the algorithm). We need to have such a global view so that we can check (or verify) some properties of the system. Global views of the system are also used to describe rewriting rules. For each of most basic actions of processes such as receipt of a marker, there are multiple rewriting rules that have been obtained by case analyzes (or case distinctions) based on predicates that are not locally observable by any process. The purpose of the case analyzes is to cover all possible situations. The case analyzes based on global predicates do not affect the action of each process designated by the system. This is because each rewriting rule whose main player is a process P can modify only the P's state, incoming channels and/or outgoing channels w.r.t. the base-state meta configuration component.

We informally reason about the algorithm to write rewriting rules. For Record of Process States, we argue the following. "If a process has already received a marker, it has already recorded its state as well. Hence, we only need to take into account the case in which a process has not yet received any markers." For **(5-1-1)** in Receipt of Markers, we argue the following. "The process has only one incoming channel, and there are no processes that have not yet completed the algorithm except for the process, which implies that the process does not have any outgoing channels." This informal reasoning can reduce the number of rewriting rules but may overlook some possible situations. Since any process of making formal models and writing formal specifications is not formal, however, any way of dosing so may overlook some possible situations. As far as we know, all we can do is to carefully make formal models and write formal specifications, and carefully validate them by some means such as animation. Since formal specifications in Maude (and any other OBJ family languages such as CafeOBJ) are executable, we can animate/execute formal specifications to validate them.

5 Model Checking of Reachability Properties

Both RP1 and RP2 can be checked with the search command. Let imc be an initial meta configuration of a system that is superimposed by the algorithm. We suppose that the system consists of n processes $\mathrm{p}(0), \ldots, \mathrm{p}(n-1)$.

The following search (called the search for snapshots) finds all states in which a snapshot has been taken:

search in EXPERIMENT :
$imc \Rightarrow^*$ start-state(SC) finish-state(FC) snapshot(SSC) MC
such that $FC \neq$ empConfig .

where EXPERIMENT is a module where the system specification module is imported and some variables such as SC, FC, SSC and MC are declared. Let m_1 be the number of the solutions to the search for snapshots.

The following search (called the search for RP1) finds all states in which a snapshot has been taken such that the snapshot SSC is reachable from the start state SC under the underlying distributed system:

search in EXPERIMENT :
$imc \Rightarrow^*$ start-state(SC) finish-state(FC) snapshot(SSC) MC
such that $FC \neq$ empConfig
$\wedge$ base-state(SC) finish-state(empConfig)
control((prog[p(0)] : notYet) ... (prog[p($n-1$)] : notYet) (consume : b))
$\Rightarrow$
base-state(SSC) finish-state(empConfig)
control((prog[p(0)] : notYet) ... (prog[p($n-1$)] : notYet) (consume : b)) .

where b is true if tokens may be consumed, and false otherwise. The rewrite expression in the condition checks if SSC is reachable from SC with the three rewriting rules "chgStt", "sndTkn" and the one for case (3-1), namely under the underlying distributed system. We need to have finish-state(empConfig) and control((prog[p(0)] : notYet) ...) in the rewrite expression to enforce using the three rewriting rules. Let m_2 be the number of the solutions to the search for RP1. If m_2 equals m_1, the algorithm enjoys RP1 w.r.t. the underlying distributed system.

RP2 can be checked likewise. All needed to do is to replace SC and SSC with SSC and FC, respectively, in the rewrite expression. This search is called the search for RP2.

Let us consider the 3-process & 4-channel system, and imc be the initial meta configuration shown in Example 2. All the searches for snapshots, RP1 and RP2 find 81,740 solutions. Therefore, the algorithm enjoys RP1 and RP2 w.r.t. the 3-process & 4-channel system. Note that the number of reachable states (meta configurations) from imc w.r.t. the 3-process & 4-channel system is 810,938.

6 Related Work

Chandy and Lamport manually prove that their algorithm enjoys RP1 and RP2 [7]. Precisely, they prove a stronger theorem that implies the two properties.

The algorithm has been analyzed based on some formal methods such as I/O automata [10] and UNITY [8].

How to analyze the algorithm based on I/O automata is as follows [10]. Each process i in an underlying system is described as an I/O automaton A_i, and a global snapshot is taken by each I/O automaton $ChandyLamport(A_i)$ that monitors A_i. What is proved is the same as what Chandy and Lamport do. The difference is to formalize the underlying system that is superimposed by the algorithm as I/O automata. But, the proof is done manually.

Chandy and Misra [8] define two properties that each snapshot algorithm should enjoy:

1. **invariant** $\langle \bigwedge x :: x.done \rangle \Rightarrow \langle \exists \mathbf{u}, \mathbf{v} :: [init]\,\mathbf{u}\,[rec] \wedge [rec]\,\mathbf{v}\,[cur] \rangle$
2. $begun \mapsto \langle \bigwedge x :: x.done \rangle$

where x is a variable used by an underlying system, $x.done$ indicates whether x has been recorded, $[init]$ is a start state, $[rec]$ is a snapshot, $[cur]$ is a finish state, $\mathbf{u}$ and $\mathbf{v}$ are computations (sequence of states), $[init]\,\mathbf{u}\,[rec]$ is a computation in which $[init]$ is the initial state and $[rec]$ is the final state, and $begun$ indicates whether a snapshot has started being taken. The second property assures that the algorithm terminates, and the first property is RP1 and RP2.

Instead of proving that the Chandy & Lamport algorithm enjoys the two properties, they provide a more fundamental rule called Rule R: when a statement in the underlying program is executed, either all variables named in the statement are recorded, or all variables named in the statement are unrecorded. They manually prove that Rule R enjoys the two properties. Precisely, they prove that Rule R enjoys a stronger version of the first property. Then, they derive the Chandy & Lamport algorithm as an implementation of Rule R.

The algorithm is manually analyzed but not mechanically in all the three cases described so far. As far as we know, there exists one case in which the algorithm is analyzed mechanically. An underlying system superimposed by the algorithm is described in Promela and analyzed with Spin[11]. One property (called the consistency property) is defined: a snapshot is consistent if it can unambiguously identify every message that has been sent as either received or as still in the channel. The underlying system used for model checking with Spin consists of two processes (a sender and a receiver) and one channel. Instead of directly checking the consistency property, the two assertions are checked when the algorithm terminates:

assert(lastSent == messageAtMarker);
assert(messageAtRecord <= messageAtMarker);

where lastSent indicates the last message that the sender put in the channel, messageAtMarker indicates the message that the receiver got from the channel just before getting a marker from it, and messageAtRecord indicates the message that the receiver got from the channel just before recording its state. The first assertion says that all messages sent by the sender before a marker have been received by the receiver, and the second assertion says that some messages denoted by messageAtRecord + 1, ..., messageAtMarker (if any) are still in the channel in the snapshot taken. RP1 and RP2 are not (at least directly) checked.

Although we know another report in which the algorithm has been mechanically analyzed, the report just mentions it[12].

7 Conclusion

We have described how to specify distributed systems superimposed by the algorithm in Maude and how to model check RP1 and RP2 with the search command, which demonstrates the power of the command, namely that more general invariant properties can be checked by the command than standard LTL and CTL

model checkers. Case analysis, which often needs to be conducted for interactive theorem proving, has played a very important role in the specification. The specification of distributed systems superimposed by the algorithm depends on neither the number of processes nor the number of channels, which demonstrates the power of Maude. Let us note that the same specification can also be described in CafeOBJ, a sibling language of Maude. The current implementation of Maude is superior to that of CafeOBJ, however, in terms of execution performance. This is why we have used Maude in the case study.

Only model checking does not let us conclude that the algorithm enjoys RP1 and RP2 for all distributed systems. One possible way to achieve this goal is to prove that if the algorithm does not enjoy RP1 and RP2 for a distributed system that consists of an arbitrary number of processes and an arbitrary number of channels, then neither does it for a smaller system that consists of a few processes and a few channels such as the 3-process & 4-channel system.

Our experience tells us that it is not straightforward to express RP1 and RP2 in existing standard temporal logics such as LTL and CTL. But, we do not have any concrete evidence of how hard it is. It would be worth investigating how hard it is.

References

1. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
2. Holzmann, G.J.: The SPIN Model Checker – Primer and Reference Manual. Addison-Wesley (2004)
3. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004)
4. Tsuchiya, T., Schiper, A.: Verification of consensus algorithms using satisfiability solving. Distributed Computing 23, 341–358 (2011)
5. An, X., Pang, J.: Model checking round-based distributed algorithms. In: 15th IEEE ICECCS, pp. 127–135. IEEE (2010)
6. Ogata, K., Futatsugi, K.: Comparison of Maude and SAL by conducting case studies model checking a distributed algorithm. IEICE Trans. Fundamentals E90-A, 1690–1703 (2007)
7. Chandy, K.M., Lamport, L.: Distributed snapshots: Determining global states of distributed system. ACM TOCS 3, 63–75 (1985)
8. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley (1988)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
10. Lynch, N.A.: Distributed Algorithms. Morgan-Kaufmann (1996)
11. Ben-Ari, M.: Principles of the Spin Model Checker. Springer (2008)
12. Konnov, I.: CheAPS: a checker of asynchronous parameterized systems. In: WING 2010. EPiC Series, vol. 1, pp. 128–129. EasyChair (2012)

Quantitative Program Dependence Graphs

Chunyan Mu

School of Computing Science, Newcastle University
Newcastle Upon Tyne, UK
chunyan.mu@ncl.ac.uk

Abstract. This paper develops a novel approach that analyses dependencies of programs in a quantitative aspect. We introduce a definition of Quantitative Program Dependence Graph (QPDG) which can be used to model a program's behaviour given spaces of inputs. The programs we consider are in a core while-language. We present the semantics for the purpose of building QPDGs. The QPDG reasons about the program's quantitative uncertainty behaviours based on a probabilistic analysis. It can be used to characterise dependence analysis of programs in a quantitative way. We also provides an optimisation of the QPDG by doing slicing in order to perform a flow analysis, e.g., how input variables at the source node might affect a given output variable at the target node and how much.

Keywords: Language, Semantics, Quantity, Dependence, Graph, Flow.

1 Introduction

Program executions contain many dependencies. Dependence analysis between entities is an important part of program analysis. Although the dependence decision made at two entities of interest is normally treated as a binary decision, the information obtained during the dependence analysis can be quantified. This paper presents an approach to dependence analysis that, rather than just approximating whether the value of one variable in a program depends on another, also quantifying to what extent a variable depends on another.

A quantitative analysis of program execution is essential to the system design process and information flow control mechanisms. A quantitative dependence analysis can be used to characterise the exact nature of dependencies between statements or variables for program analysis. In this paper, we propose a representation of the program dependence graph that can be used to capture the properties of dependence relations between variables by executing the program. By using the idea of information theory [19], we compute the quantity of the *interference* introduced by data dependence among *variables* at points of interest. In the security community, it can be used to capture the information flow from sensitive inputs to public outputs by executing the program, and therefore we can calculate how much information can be learned about the security input operation by observing the public output. Intuitively, information flow

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 103–118, 2012.

in programs is naturally related to the *interference* between variables, which is naturally related to the *dependence* relations between variables given program points. Dependency and interference play a key role in designing suitable abstractions or in partitioning complex systems into simpler ones. Identifying and measuring dependency and interference is useful in many fields, particularly security, program analysis and verification. We therefore motivate our idea to provide a quantified dependence analysis for programs.

Related Work. There are a number of techniques that have been developed for dependence representations of programs, e.g., [6,14,20,8,9]. Ottenstein and Ottenstein [17] use the PDG for static slicing of single-procedure programs. They define slicing as a reachability problem in a dependence graph representation of a program. Jackson and Rollins [8] introduce a PDG that is distinguished by fine-grained dependences between individual variables defined or used at program points. The quantified program dependences investigated in this paper is more relevant to the underline model of dependences between variables discussed in [8]. They are difficult to automatically capture quantified dependence between variables. [1] presented a probabilistic program dependence graph model based on the PDG called PPDG. The PPDG captured the statistical dependences among *program statements* and enabled the use of probabilistic reasoning to analyse program behaviors. Two applications of the PPDG to fault localization and fault comprehension were presented by simple calculations on the information of suspicious statements of the program. Although a lot of efforts on the development of program dependence graphs, to the best of our knowledge there is no research conducted to a *quantitative* version of dependence analysis between *program variables*, which is our main contribution in this paper. Our work can also relate to the topic of quantitative information flow analysis for programs including [5,4,11,7,2,16,15,3,13], and fault localisation based on dependencies in programs including [10,12,22,1], as an application.

Contributions. We consider a simple programming language, give the syntax and semantics of the language, describe the QPDG representation for this language, and construct the dependencies between variables from the syntax of the program statements. By semantically interpreting the concepts of control and data dependence, we derive a denotational definition of the control and data demand generated by variables at program points of interest. We present the definition of QPDG, which can be used to provide a *quantitative* analysis of dependencies between *program variables* by executing the program. Furthermore, we give a simple algorithm in order to build a reduced QPDG which can be used to perform quantitative analysis of unified dependences and information flow. Intuitively, we are particularly interested in which variables at a source node might affect a given variable at a target node and how much in information bits. We also discuss the possible applications of the reduced QPDG to flow measurement in the security community, and show that simple and intuitive computation can be obtained.

Outline. In section 2, we recall some definitions from the literature of program dependence graphs, random variables and information entropy. Section 3 defines

the syntax and semantics for a simple while language and semantically describes the QPDG for this language. Section 4 introduces our definition of QPDG to describe quantified dependencies between program variables within program operations. 5 presents an algorithm for building a reduced QPDG to provide a more efficient quantitative dependence analysis and suggest a possible application to information flow measurement. The final Section summarises our work and presents directions for future research.

2 Preliminaries

In this section, we briefly review some definitions in the relevant background including program dependence graph, random variables and programs.

2.1 Program Dependence Graph

Many analyses and transformations of programs are based on dependence relations which are normally represented by program dependence graph (PDG) [6]. The PDG plays an important role in expressing the essential dependencies of atomic program operations. We use the dependence graph as our intermediate representation basis. Intuitively, the dependence graph can be viewed as a data structure in which edges represent dependencies between operations. Dependence graphs integrate data and control dependence information into a single structure, making efficient algorithms for program analysis. A dependence graph consists of a set of nodes representing functional operators and a set of edges representing the *dependencies* and *precedence relations* that exist between those operations.

In this paper, we develop a representation of PDG that can be used to capture the *quantified* dependencies between *variables* rather than just the statements themselves. For instance, we are interested in how a specific output variable at a target node might be affected by a given input variable at a source node.

2.2 Random Variables and Programs

There is a clear connection between the notion of probability space, information theory, and dependence & interference extent in a program. A probability space [18] is a measure space such that the measure of the whole space is equal to one. Specifically, a probability space is a triple $(\Omega, \mathcal{B}, \mu)$: Ω is the sample space, an arbitrary non-empty set; the σ-algebra $\mathcal{B} \subseteq 2^{\Omega}$ is a set of subsets of Ω, called events, such that contains the empty set: $\emptyset \in \mathcal{B}$; is closed under complements: if $A \in \mathcal{B}$, then also $(\Omega \setminus A) \in \mathcal{B}$; and is closed under countable unions: if $A_i \in \mathcal{B}$ for $i = 1, 2, \ldots$, then also $(\bigcup_i A_i) \in \mathcal{B}$; the measure of entire sample space is equal to one: $\mu(\Omega) = 1$. In order to investigate the quantitative dependence analysis between program variables for programs, we consider *program variables* as *random variables* and *semantic functions* as *measurable functions*. A random variable is a measurable function from a probability space into a measurable space. Let X

and Y are probability spaces, mapping $f : X \to Y$ is a measurable function if for all W measurable in Y, $f^{-1}(W)$ is measurable in X. The denotational semantics of commands is a mapping from the set X of possible environments before the commands into the set Y of possible environments after the commands.

2.3 Information Entropy

Information theory [19] introduced the definition of *entropy*, $\mathcal{H}$, to measure the *average uncertainty* in random variables. Consider a program as a state transformer, random variable X is a mapping between two states which are equipped with distributions. Let $p(x)$ denote the probability that X takes the value x, then the *entropy* $\mathcal{H}(X)$ of discrete random variable X is defined as: $\mathcal{H}(X) = \sum_x p(x) \log_2 \frac{1}{p(x)} = -\sum_x p(x) \log_2 p(x)$. The concept of *mutual information* is a measure of the amount of information that one random variable contains about another one, i.e., shared information. It implies the reduction in the uncertainty of one random variable due to the knowledge of the other. It therefore reasonably suggests the quantity of *dependencies* between random variables, i.e., knowledge of one may change the information about another. Let $p(x, y)$ denote the joint distribution of $x \in X$ and $y \in Y$, the *mutual information* between X and Y, $\mathcal{I}(X;Y)$, is given by: $\mathcal{I}(X;Y) = \sum_x \sum_y p(x,y) \log_2 \frac{p(x,y)}{p(x)p(y)}$. The conditional version of mutual information can be considered as the reduction in the uncertainty of X due to knowledge of Y when Z is given by: $\mathcal{I}(X;Y|Z) = \mathcal{H}(X|Z) + \mathcal{H}(Y|Z) - \mathcal{H}(X,Y|Z)$.

3 The Semantics and Program Dependences

This section presents the language and semantics aimed at semantically expressing the dependency between program variables. In order to perform a *quantitative* analysis of program dependencies between *variables* for programs, we consider the program variables as random variables which take probability spaces as mapped values rather than single values, i.e., there are probability distributions on the state space rather than just values.

3.1 The Abstract Syntax

To simplify our presentation and focus on the problem of quantified dependence analysis of programs, we consider a simplified language. We present the syntax of the core language in Table 1, where s ranges over statements, l denotes a label, x ranges over a set of variables, b ranges over a set of boolean expressions, and e ranges over a set of arithmetic expressions.

3.2 Semantic Domains

We have the following semantic domains:

Table 1. The abstract syntax of the language

$s \in \textsc{Stmt}$ $l \in \textsc{Lab}$ $x \in \textsc{Ide}$ $e \in \textsc{Exp}$ $b \in \textsc{BExp}$ $n \in \textsc{Num}$
$s ::= \texttt{input}(x) \mid \texttt{skip} \mid x := e \mid s_1; s_2 \mid \texttt{if}\ b\ \texttt{then}\ s\ \texttt{else}\ s \mid \texttt{while}\ b\ \texttt{do}\ s$
$e ::= x \mid n \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid e_1 \% e_2$
$b ::= \neg b \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 = e_2$

$$\textsc{Val} \stackrel{\text{df}}{=} (\Omega, \mathcal{B}, \mu) \qquad \sigma \in \textsc{Store} \stackrel{\text{df}}{=} \textsc{Ide} \mapsto \textsc{Val}$$
$$\delta \stackrel{\text{df}}{=} \mathcal{P}(\textsc{Ide}) \qquad \Delta_l \in \textsc{Dep} \stackrel{\text{df}}{=} \mathcal{P}(\textsc{Ide} \mapsto \delta \cup \textsc{Val})$$

where Val is a probability space $(\Omega, \mathcal{B}, \mu)$; $\sigma \in$ Store is a state which maps Ide to probability spaces; δ denotes the set of program variables which a given expression evaluation depends on; Δ_l denotes the a superset of the dependencies between variables for a statement with label l each of which can be treated as a function of type $x \mapsto \delta \cup$ Val, where $x \in$ Ide: variable x depends on a set of variables in δ or Val by executing the statement with label l. Intuitively, δ can be viewed as a set of place-holders in an expression whose values are substitute to evaluate the expression. Hence, the value of expression depends on the variables within it.

3.3 The Semantics

Denotational semantics is closely related to dependence analysis. We formulate the probabilistic denotational semantics so that the dependencies can be extracted and recorded for the program. The assignment is ignored unless it affects the visible contents of the program store. A definition of our denotational semantics is given in Table 2. For clear cases, we use $\mu_{\textsc{Ide}}$ to denote the probability space of Ide in stead of writing $(\Omega, \mathcal{B}, \mu)$.

Table 2. Denotational Semantics of Programs

$$\begin{aligned}
[\![s]\!] &: \textsc{Stmt} \to (\textsc{Store} \times \textsc{Dep} \to \textsc{Store} \times \textsc{Dep}) \\
[\![e]\!] &: \textsc{Exp} \to (\textsc{Store} \to \textsc{Val}) \to \delta \\
[\![b]\!] &: \textsc{BExp} \to (\textsc{Store} \to \textsc{Store}) \to \delta
\end{aligned}$$

$$\begin{aligned}
[\![\texttt{input(x)}]\!]_l\ \sigma_l \Delta_l &\stackrel{\text{df}}{=} \sigma_{l'}(x \mapsto \mu_x) \Delta_{l'}(x \mapsto \mu_x) \\
[\![x := e]\!]_l\ \sigma_l \Delta_l &\stackrel{\text{df}}{=} \lambda W. \sigma_l([\![x := e]\!]_l^{-1}(W)) \Delta_{l'}(x \mapsto \delta(e)) \\
[\![s_1; s_2]\!]_l\ \sigma_l \Delta_l &\stackrel{\text{df}}{=} [\![s_2]\!] \circ [\![s_1]\!] \sigma_l \Delta_l \\
[\![\texttt{if b then s}_1\ \texttt{else s}_2]\!]_1\ \sigma_l \Delta_l &\stackrel{\text{df}}{=} \Delta_{l'}(\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s_1)\}) [\![s_1]\!] \circ [\![b]\!] \sigma_l \Delta_l \bigcup \\
&\quad\ \Delta_{l'}(\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s_2)\}) [\![s_2]\!] \circ [\![\neg b]\!] \sigma_l \Delta_l \\
[\![\texttt{while b do s}]\!]_1\ \sigma_l \Delta_l &\stackrel{\text{df}}{=} \Delta_{l'}(\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s)\}) \\
&\quad\ [\![\neg b]\!](\lim_{n \to \infty}(\lambda \sigma_l' \Delta_l'. \sigma_l \Delta_l \bigcup [\![s]\!] \circ [\![b]\!](\sigma_l' \Delta_l'))^n)(\lambda X. \bot)) \\
&\quad\ \text{where, } [\![b]\!] \sigma_l = \lambda X. \sigma_l(X \cap \mathcal{M}_b)
\end{aligned}$$

An arithmetic expression is a function $[\![e]\!] : \sigma \mapsto \textsc{Val}$ by using two's-complement interpretations of $+, -, *, /$ as standard. Let $\delta(e)$ denote a set of variables calculating e depends on, i.e., the value of $[\![e]\!]$ depends at most on the variables in $\delta(e)$. A boolean expression is interpreted as a function $[\![b]\!] : \sigma \to \sigma$. The function defines the part of the store matched by the condition b. In addition, the value of $[\![b]\!]$ depends at most on the variables in $\delta(b)$. For example, assume the current store is: $\sigma\left(x \mapsto \left[0 \mapsto \frac{1}{4}, 1 \mapsto \frac{1}{4}, 2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\right]\right)$. Consider the expression b is $[\![x > 1]\!]$, we have $\delta(b) = \{x\}$ and the updated store under condition b as: $\sigma\left(x \mapsto \left[2 \mapsto \frac{1}{4}, 3 \mapsto \frac{1}{4}\right]\right)$. A command is interpreted as a function: $[\![s]\!] : \sigma \times \Delta \to \sigma \times \Delta$.

The semantics is a store transformer that carries program dependencies. The meaning of a statement s is not a transformer of the entire input store σ but a transformer only the part of the store consisting of the variables that are defined in s. For a given statement s, $\textsc{Def}(s)$ returns the superset of all variables that may be assigned to. Given a label l, Δ_l returns the superset of maps: $\mathcal{P}(\textsc{Ide} \to \delta \cup \textsc{Val})$ which constitute dependencies for the statement with that label. Specifically,

- *Random input* `input(x)` assigns a probability space μ_x to x, and builds a dependence relation such that variable x depends on the input space: $\textsc{Val}_x = \mu_x, \quad \sigma(x \mapsto \textsc{Val}_x), \quad \Delta(x \mapsto \textsc{Val}_x)$.
- *Assignment* updates the store such that the state of the assigned variable x is updated to become that of expression e, and updates the dependence such that $\Delta(x \mapsto \delta(e))$. The distribution transformation function for assignment is presented by using an inverse image: $\lambda W.\sigma([\![x := e]\!]_l^{-1}(W))$ to keep the measurability of the semantic function. For all measurable $W \in \sigma'$, $[\![x := e]\!]_l^{-1}(W)$ is measurable in σ, where σ and σ' denote the state before and after the assignment operator.
- The distribution transformation function for the *sequential composition operator* is obtained via functional composition:

$$[\![s_1; s_2]\!]_l \sigma_l \Delta_l = [\![s_2]\!] \circ [\![s_1]\!] \sigma_l \Delta_l.$$

Note that Δ_l is a binary relation on variables. The relational composition is considered as follows: assume δ_x denotes the set of variables that x depends on by executing the statement s_1, δ_y denotes the set of variables on which y depends by executing the statement s_2, assume $x \in \textsc{Def}(s_1)$, and $y \in \textsc{Def}(s_2)$, we compute:

$$\begin{aligned}
&[\![s_1]\!]\Delta_l = \Delta'_l(\{x \mapsto \delta_x \mid x \in \textsc{Def}(s_1)\}) \\
&[\![s_2]\!]\Delta'_l = \Delta_{l'}\big(\{x \mapsto \delta_x \mid x \in \textsc{Def}(s_1) \wedge x \notin \textsc{Def}(s_2)\} \cup \\
&\qquad\qquad \{y \mapsto (\lambda x.\delta_y)\delta_x \mid y \in \textsc{Def}(s_2),\ x \in \textsc{Def}(s_1)\}\big),
\end{aligned}$$

where we use $(\lambda x.\delta_y)\delta_x = [\delta_x/x]\delta_y$ to indicate that all occurrences of x in δ_y are substituted by δ_x. We therefore write:

$$[\![s_1; s_2]\!]_l \Delta_l = ([\![s_2]\!] \circ [\![s_1]\!])_l \Delta_l$$

$$= \Delta_{l'}\big(\{x \mapsto \delta_x \mid x \in \textsc{Def}(s_1) \wedge x \notin \textsc{Def}(s_2)\} \quad \cup \\ \{y \mapsto (\lambda x.\delta_y)\delta_x \mid y \in \textsc{Def}(s_2),\ x \in \textsc{Def}(s_1)\}\big).$$

Hence the $\circ$ operator for sequential statements ensures that the dependencies which come from previous statements are recorded; and only dependencies for variables that are defined in the last statement are kept.

– The boolean function $[\![b]\!]$ for *boolean test* b defines the part of the space defined in the current store and matched by the condition b: $[\![b]\!]\sigma = \lambda X.\sigma(X \cap \mathcal{M}_b)$. The boolean test causes the space to split apart, $X \cap \mathcal{M}_b$ denotes the part of the space which makes the boolean test b to be true. We use $\delta(b)$ to denote the set of variables that are required for evaluating b.
– A *Conditional statement* is executed with the conditional probability distributions for either the *true* branch or *false* branch:

$$\Delta_l(\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s_1)\})[\![s_1]\!] \circ [\![b]\!]\sigma_l\Delta_l \cup \\ \Delta_l(\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s_2)\})[\![s_2]\!] \circ [\![\neg b]\!]\sigma_l\Delta_l$$

where $\Delta_l(\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s_1)\})$ means that, $\forall x \in \textsc{Def}(s_i)$, $i = 1, 2$, we update the dependence relation of x by union $\delta(b)$, where $\textsc{Def}(s_i)$ denotes the set of variables are assigned to during the evaluation of statement s_i, δ_x denotes the set of variables that x depends on after executing $[\![s_i]\!]$, i.e., $[\![s_i]\!] \circ [\![b]\!]\Delta_l$. For instance, consider a piece of program with an if statement:

$$[\![\texttt{if (x > 1) then y := y - 1 else y := y + 1;}]\!]_1.$$

Assume the space of x, y at the beginning of the program is written as:

$$\sigma_l \left[x \mapsto \left(0 \mapsto \frac{1}{4} \quad 1 \mapsto \frac{1}{4} \quad 2 \mapsto \frac{1}{4} \quad 3 \mapsto \frac{1}{4}\right), \quad y \mapsto (0 \mapsto 1)\right].$$

After executing the if statement, according to the semantics, the store is updated as:

$$\sigma_{l'}\left[x \mapsto \left(0 \mapsto \tfrac{1}{4}\ 1 \mapsto \tfrac{1}{4}\ 2 \mapsto \tfrac{1}{4}\ 3 \mapsto \tfrac{1}{4}\right), \quad y \mapsto \left(-1 \mapsto \tfrac{1}{2}\ 1 \mapsto \tfrac{1}{2}\right)\right],$$

and the dependence relation is updated as: $\Delta_{l'}(y \mapsto y \cup \delta(b)) = \Delta_{l'}(y \mapsto \{x, y\})$ i.e.,

$$[\![\texttt{if (x > 1) then y := y - 1 else y := y + 1;}]\!]_1\sigma_1\Delta_1 \\ = \sigma_{l'}\left[x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4}\ 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4}\ 3 \mapsto \frac{1}{4} \end{pmatrix}, \quad y \mapsto \begin{pmatrix} -1 \mapsto \frac{1}{2} \\ 1 \mapsto \frac{1}{2} \end{pmatrix}\right] \quad \Delta_{l'}(y \mapsto \{x, y\}).$$

– The *loop* statement can be represented as an infinite nested if statement, the state space with distribution μ defined in the current store goes around the loop, and at each iteration, the remaining part that makes test b *false* breaks off and exits the loop, while the rest of the space goes around again. The output distribution $[\![\texttt{while b do s}]\!]_1\sigma_1$ is thus the sum of all the partitions that finally find their way out. For non-termination cases, we consider the quantity of the dependence is 0. The dependence representation, $\{x \mapsto \delta_x \cup \delta(b) | x \in \textsc{Def}(s)\})$, defines the dependencies introduced by the loop.

4 Quantitative Program Dependence Graph

This section introduces our definition of QPDG to describe quantified dependencies between program variables within program operations based on the semantics defined in Section 3. A program dependence graph defines a partial ordering on the statements and the program performance to preserve the semantics of the original program. A quantified program dependence graph consists of a set of nodes representing atomic computations (assignments and predicate expressions) with current stores regarding to σ and a set of edges representing the dependencies regarding to Δ. We adapt the nodes to accommodate the current *store* and *dependencies* between variables introduced by the statement in the node. The dependencies relate to definitions and uses of program variables rather than the whole program statements. The set of edges contains *data dependence* edges and *control dependence* edges.

There are some differences between our QPDG and the standard PDG. First, rather than considering dependencies between statements, we consider the dependencies between variables. Second, the node of the QPDG is labelled by program *statement blocks*: $\textsc{Block} \in \mathcal{P}(\textsc{Stmt})$. Third, each node accommodates the stores which record the states of each variable obtained from the stores σ and the dependence relations defined in Δ given by the semantic functions. Intuitively, the node of the QPDG can be viewed as a *transformation box*, which accommodates a statement block with label l, a set of entry ports and a set of exit ports, and a set of directed internal edges between entry ports and exit ports of the node. Each entry port stores the state of a "*will-be-used*" variable before executing the statement block. Each exit port stores the state of each *defined* variable after executing the statement block. Note that nodes can be nested for the case of nested if statements and loops. Fourth, directed internal edges between entry ports and exit ports of a node are used for connecting multiple pairs of variables according to their dependencies obtained from Δ within the node. The dependence (denoted by an internal edge) between each pair is introduced by executing the statement block accommodated at this node. Furthermore, external dependence edges are essentially data dependence edges between two nodes. An external dependence edge starts from an exit port (accommodates a defined variable and its state) of a node and ends at an entry port (accommodates a used variable and its states) of another node. It implies that there is a precedence relation and a data dependence relation between these two variables at these two nodes. In addition, we define the quantity of a dependence edge based on conditional mutual information which will be discussed later. The value of the quantified dependence edge indicates the content of the dependency between a pair of program variables. We now present the definition of quantitative program dependence graph as follows.

Definition 1. *A quantitative program dependence graph (QPDG) is defined as a pair:* $QPDG = (N, E_x)$*, where,*

(i) $N = \mathcal{P}(\textsc{Block}, P_{\texttt{entry}}(\sigma), P_{\texttt{exit}}(\sigma), E_i(\Delta))$ *is a set of tuples of executable statement boxes each of which accommodates a statement* Block*, a set of*

entry ports $P_{\boldsymbol{entry}}(\sigma)$ *and a set of exit ports* $P_{\boldsymbol{exit}}(\sigma)$*, and directed internal dependencies edges denoted by* $E_i(\Delta)$.
Each entry port accommodates a will-be-used variable with its state defined in σ *before executing the statement* BLOCK.
Each exit port accommodates a defined variable with its state defined in σ *after executing the statement block* BLOCK.
Directed internal edges E_i *between entry ports and exit ports connects multiple pairs of variables which can be extracted from their dependencies given in* Δ *defined by the semantics. Specifically,* E_i *is a set of directed edges inside the nodes. Each edge* $e \in E_i$ *is a tuple* $(p_{\boldsymbol{entry}}, p_{\boldsymbol{exit}}, q_e)$*, where* $p_{\boldsymbol{entry}} \in P_{\boldsymbol{entry}}$*,* $p_{\boldsymbol{exit}} \in P_{\boldsymbol{exit}}$*, and* q_e *denotes the quantity of* e *defined by* $q_e = \mathcal{I}(X; Y'|Z)$*, where* X *denotes the random variable located at* $P_{\boldsymbol{entry}}$*, and* Y' *denotes the random variable located at* $P_{\boldsymbol{exit}}$*, and* Z *denote the joint random variable of a set of variables located in* $Q_{\boldsymbol{entry}} = P_{\boldsymbol{entry}} \setminus p_{\boldsymbol{entry}}$*, if there exists an internal edge* $e' \in E_i$*,* $q_{\boldsymbol{entry}} \in Q_{\boldsymbol{entry}}$*, and* $e' \neq e$ *where* $e' = (q_{\boldsymbol{entry}}, p_{\boldsymbol{exit}}, q_{e'})$*. Intuitively, each internal edge connects an entry port which accommodates a variable will-be-used in the node and an exit port which accommodates a defined variable in the node. The internal flow edges* E_i *imply the dependencies introduced by the statement block of the nodes.*
Note that the store σ *(defined in the semantics) is a set of maps from the program variables to their probability spaces, and* Δ *denotes a set of dependencies between variables introduced by the statement block.*

(ii) E_x *is a set of directed external edges among the nodes. Each edge of* E_x *starts from an exit port of one node* m *and ends at an entry port of another node* n*. Specifically, a defined variable in node* m *will be used in node* n*, where* $m, n \in N$*. Intuitively, a data flow edge between nodes actually connects programs variables and implies a data flow dependence.*

Example 1. Consider a piece of program as follows.

```
l1: input(x);
l2: y:=0;
l3: if (x % 2 ==0) then y:= y-1 else y:=y+1;
```

According to the intuitive description of the QPDG above, we have three nodes (transformation boxes). The first two denote the random input and assignment respectively, and the third one denotes the if statement. Assume that the state of x introduced by `input(x)` is: $\sigma_{l_1}\left[x \mapsto \left(0 \mapsto \frac{1}{4}\ 1 \mapsto \frac{1}{4}\ 2 \mapsto \frac{1}{4}\ 3 \mapsto \frac{1}{4}\right)\right]$. We present the elements of the graph for this example in Table 3 and the QPDG in Fig. 1 to show some intuitions of the graph.

Intuitively, each node is a transformation box with two sides of "ports": entry ports and exit ports. Each entry port accommodates a "will-be-used" variable with its state before executing the statement block. Each exit port accommodates a "defined variable" with its state after executing the statement block. If there is an internal connection between an entry port and an exit port within the box

Table 3. A simple example of QPDG: the nodes

semantic blocks (nodes)	entry ports	exit ports	dependencies Δ
`l1: input(x)`	VAL_x	$x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} & 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} & 3 \mapsto \frac{1}{4} \end{pmatrix}$	$x \mapsto \text{VAL}_x$
`l2: y:=0`	VAL_y	$y \mapsto (0 \mapsto 1)$	$y \mapsto \text{VAL}_y$
`l3: if (x%2 == 0) then y:=y-1 else y:=y+1`	$x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} & 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} & 3 \mapsto \frac{1}{4} \end{pmatrix}$ $y \mapsto (0 \mapsto 1)$	$y \mapsto (-1 \mapsto \frac{1}{2}\ 1 \mapsto \frac{1}{2})$	$y \mapsto \{x, y\}$

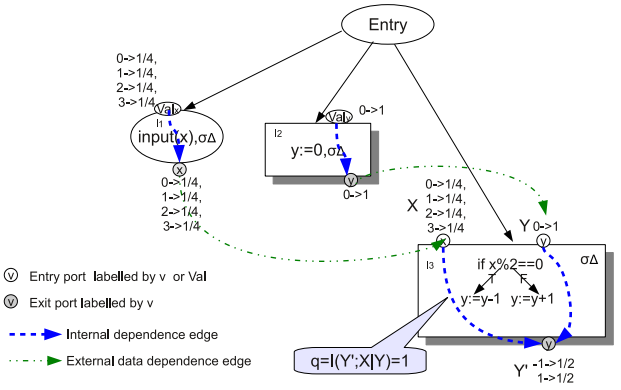


Fig. 1. A simple example of QPDG: the graph

then there is a dependency between them which is introduced by the statement of this node. For instance, the internal dependence edges within the node with label l_3 means that the definition of y in the if statement depends on variables x and y according to the semantics. Let $q(\Delta_{l_3}(y \mapsto x))$ denote the quantity of the dependence between the variable y and the variable x. Let X, Y, Y' denote the random variables of the variable x at the entry port before executing the program, program variable y at the entry port (before executing the program), and program variable y at the exit port (after executing the program). We have: $q(\Delta_{l_3}(y \mapsto x)) = \mathcal{I}(Y'; X|Y) = 1$. This also meets our intuition: the space of Y' depends on the last bit of X (0 to be even and 1 to be odd regarding to the control dependence introduced by the boolean test `x%2 == 0`) . In addition, the external data dependence edge between the node of `input(x)` and the node of the if statement suggests that the will-be-used variable x at the if statement node (entry port) depends on the defined variable at the node of `input(x)` (exit port).

Example 2. Consider an example program P presented in Fig. 2. Assume the distribution of possible values of x produced by statement `input(x)` is:

$$\mu_x : x \mapsto \left(0 \mapsto \tfrac{1}{4}\ 1 \mapsto \tfrac{1}{4}\ 2 \mapsto \tfrac{1}{4}\ 3 \mapsto \tfrac{1}{4}\right)$$

```
l1:  input(x);
l2:  z := 1;
l3:  y := 0;
l4:  while (x>y) {
       y := y+1;
       z := z*y;
     }
l5.  output(y);
```

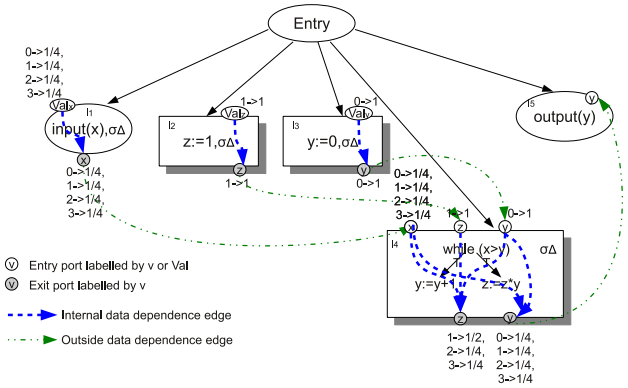


Fig. 2. Program P and its QPDG

According to the semantic function defined in Table 2, the transformation of program P can be written as:

$$[\![\texttt{input}]\!]_{l_1}\sigma_{l_1}\Delta_{l_1} = \sigma_{l_2}\left[x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} & 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} & 3 \mapsto \frac{1}{4} \end{pmatrix}\right] \quad \Delta_{l_2}(x \mapsto \mathrm{VAL}_x)$$

$$[\![z := 1]\!]_{l_2}\sigma_{l_2}\Delta_{l_2} = \sigma_{l_3}\left[x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} & 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} & 3 \mapsto \frac{1}{4} \end{pmatrix},\ z \mapsto 1 \mapsto 1\right] \quad \Delta_{l_3}(z \mapsto \mathrm{VAL}_z)$$

$$[\![y := 0]\!]_{l_3}\sigma_{l_3}\Delta_{l_3} = \sigma_{l_4}\left[x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} & 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} & 3 \mapsto \frac{1}{4} \end{pmatrix},\ z \mapsto 1 \mapsto 1,\ y \mapsto 0 \mapsto 1\right] \Delta_{l_4}(y \mapsto \mathrm{VAL}_y)$$

$$[\![\texttt{while}]\!]_{l_4}\sigma_{l_4}\Delta_{l_4} = \sigma_{l_5}\left[x \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} \\ 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} \\ 3 \mapsto \frac{1}{4} \end{pmatrix},\ z \mapsto \begin{pmatrix} 1 \mapsto \frac{1}{2} \\ 2 \mapsto \frac{1}{4} \\ 3 \mapsto \frac{1}{4} \end{pmatrix},\ y \mapsto \begin{pmatrix} 0 \mapsto \frac{1}{4} \\ 1 \mapsto \frac{1}{4} \\ 2 \mapsto \frac{1}{4} \\ 3 \mapsto \frac{1}{4} \end{pmatrix}\right]$$
$$\Delta_{l_5}(y \mapsto \{x, y\}, z \mapsto \{y, z\})$$

Note that the while loop introduces a set of data flow dependencies. For instance, there is an edge between statement block l_1 and l_4 which implies that the defined variable x at point l_1 affects the value of the boolean test of the while loop at point l_4; and there is an edge between point l_3 and l_4 which implies that the defined variable y at point l_3 used by the statement at l_5. A translation of the P into the QPDG is presented in Fig. 2. Let us consider the node of while loop denoted as n_{l_4} as an example. Clearly:

$$n_{l_4} = \Big([\![\texttt{while}\ (x > y)\ \texttt{do}\ y := y + 1;\ z := z * y;]\!]_{l_4}, P_{\texttt{entry}}, P_{\texttt{exit}}, \{e_i | 1 \le i \le 4\}\Big)$$

where, $P_{\texttt{entry}} = \{x, z, y\}$, $P_{\texttt{exit}} = \{x, z, y\}$.

Consider two internal dependence edges of interest for an example: $e_1 = (x, z, \frac{2}{3})$ and $e_2 = (x, y, 2)$. Note that the quantity of internal dependence edge e_i is computed by conditional mutual information as discussed above. For example, the quantity of the dependence edge e_1 and e_2 of interest can be computed by: $q_{e_1} = \mathcal{I}(Z'; X|Y) = \mathcal{I}(Z'; X) = \frac{2}{3}$ and $q_{e_2} = \mathcal{I}(Y'; X|Y) = \mathcal{I}(Y'; X) = 2$, where X, Y, Y', Z, Z' denote the random variables of the program variable x before

executing the loop block, program variable y before and after executing the loop block, program variable z before and after executing the loop block.

5 Reducing QPDG by Slicing for Flow Analysis

We introduce an algorithm to build reduced QPDG in this section. Intuitively, the reduced QPDGs extract the part of the program that introduces dependences from the source node of interest to the target node of interest.

The dependence we concern for flow analysis relates a variable at one program point to a variable at another, e.g., we are particularly interested in whether or not an output variable at a target node depends on an input variable at a source node and how much. We introduce a reduction on the QPDG by doing slicing [21] on it in order to extract the part of a program (the slice) which is relevant to a subset of the program dependence behaviour of interest. Specifically, given the target node and source node of interest, we perform a backward slicing to chop out the nodes and data flow components without affecting the execution of the target node. In this way, we chop the program dependence graph to extract the sub-graph of the program that affects the values of the target node and filters the components of the graphs that do not affect the values of it. Given a source node, if the source node is not included in the chopped graph, the target node is not affected by the source node and hence there is no interference between the output variable at the target node and the input variable defined at the source node.

5.1 Reducing QPDG

Slicing is a program analysis technique developed by Weiser [21] for imperative languages. It contains all parts of a program that affect a program variable v at a statement labelled by l. The pair (v, l) is called slicing criterion, and a slice is computed regarding to a slicing criterion. In this section, we introduce a reduction on the QPDG based on the idea of slicing. A slice w.r.t. any variable that is either defined or used at a program point can be extracted directly from the graph. This enables us to determine more accurately which variables are responsible for the inclusion of a particular statement in a slice. We first present the definition of the reduction on the QPDG as follows.

Definition 2. *A reduced graph, RG, is an abstraction of the original QPDG G such that:*

(i) RG is an abstract graph of the original graph, written as: $RG \succeq G$.
(ii) only the nodes of interest are included in the reduced graph RG, i.e., $N^R \subseteq N$, *where* N^R *and* N *denote the set of the nodes of RG and G respectively;*
(iii) only the edges of interest are included in the reduced graph RG: $E_x^R \subseteq E_x$ *and* $\forall i \in N^R \wedge i \in N,\ E_i^R \subseteq E_i$, *where* E_x^R *and* E_x *denote the set of external dependence edges of RG and G,* E_i^R *and* E_i *denote the set of internal dependence edges of any node i in RG and the corresponding ones in G;*

(iv) let f_G and f_{RG} be the space transformation function of the G and RG, we have: $f_{RG} \succeq f_G$ i.e., the reduced graph computes the same space transformation of interest for the original graph does.

Specifically, a backward slice is defined to contain the statements of the program that are affecting the slicing criterion. We reduce the QPDG based on backward slicing. The backward slied QPDG filters the nodes and edges that are not affecting the target variable at the target point. A backward slice consists of all the nodes that affect the state of a given variable at a given target point in the program. In what follows, we present the description to build a backward slied QPDG.

1. Given the target variable and the target node, the source variable and the source node, assume the original QPDG is denoted by G.
2. By doing a backward slicing from the target node, we obtain the sets of statements of the program that influence the output variable at the target node. In particular, we follow the edges backwards performing a simple graph traversal and marking nodes encountered on the way. Let RG^{ι} denote the graph obtained after doing the backward slice.
3. Find all the variables at their definition points, denoted by: $\Sigma_0 = \{(z, n) | n \in N, z \in \text{IDE}\}$, which affect the final value of the target variable at the target point, i.e., a definition of variable z at node n reaches the target node.
4. If the source node is not included in the obtained graph RG^{ι}, then there is no dependence or interference between the definition of the source variable at the source node and the target variable at the target node. The algorithm is terminated here and the quantity of the dependency between the source variable at source and the target variable at target is 0.
5. Otherwise, we can compute the quantity of the dependency denoted by $d(x \rightsquigarrow y)$ between the relevant random variable defined at source node denoted by X and the relevant random variable used at the target node denoted by Y': $d(x \rightsquigarrow y) = \mathcal{I}(Y'; X|Z)$, where Z denote the joint random variable of all the variables at their definition points i.e.,

$$\Sigma_0 \setminus X = \{(z, n) \mid n \in N, (z, n) \in \Sigma_0 \wedge z \neq x\}.$$

The relationship between the backward sliced QPDG RG^{ι} by performing backward slicing discussed above and the corresponding QPDG G is described in the following theorem.

Theorem 1. *RG^{ι} is a reduced graph of G which makes same state and dependence relation of the selected variable at the selected point.*

Proof. By definition, all statements in RG^{ι} by performing backward slice make a difference in the results at the selected point. These statements in the slice either directly or indirectly affect the results. First, for the case of directly affecting the results: nodes accommodate these statement BLOCKs are included in the corresponding QPDG G and backward sliced QPDG RG^{ι} according to Definition 1 (i) and the description of building an RG^{ι}. Specifically, nodes of

RG^{ι}, $N^R = \{N_1^R, \ldots, N_x^R, \ldots, N_y^R\}$, are a subset of nodes of the corresponding G, $N = \{N_1, \ldots, N_x, \ldots, N_y, \ldots, N_n\}$, i.e., $N^R \subseteq N$, $N_x^R = N_x$, $N_y^R = N_y$, where N_y^R and N_y denote the target nodes of the RG^{ι} and G respectively, N_x^R and N_x denotes the source nodes of the RG and G respectively. Second, the sequential operators in the semantics ensures the transitivity of the dependencies. Therefore, for the case of indirectly affecting the results: external edges and corresponding nodes are included in the corresponding QPDG G and RG^{ι} according to Definition 1 (ii) and the description of building an RG^{ι}. Specifically, external edges of RG^{ι}, are a subset of external edges of the corresponding G, i.e., $E_x^R \subseteq E_x$. Therefore, according to Definition 2, RG^{ι} is a reduced QPDG of G which makes same results of the output variable at the selected point.

5.2 Applying to Quantified Flow Analysis

Information flow analysis is an aspect of computer security concerned with how security information is allowed to flow through a computer system. Consider the following examples, which shows that the dependence relations cause secure information flow to be violated during the execution of the programs:

```
1) l:=h+5;
2) if (h % 2 == 0) then l:=0 else l:=1;
3) l:=0; while (h>l) l:=l++;
```

where l is a low level variable, h is a high level variable. It is clear that, the *assignment* command contains a *data dependence* between h and l and thus causes the *explicit flow*, both the *if statement* and the *loop* contains *control dependence* between h and l which causes the *implicit flows*. Formally, secure information flow of a terminating program can be described as follows.

Definition 3. *Given a terminating program P with high variables $H = h_1, \ldots, h_n$ and low variables $L = l_1, \ldots, l_n$, P is secure if and only if the values of L at the point that P terminates are independent of the initial values of H.*

Note that the dependence relations between program variables with different security types reduces the uncertainty of secure information and causes the information flow. The reduced QPDG discussed in this paper can be used to provide a quantified flow analysis. Intuitively, given two points of interest (high input and low output), the chopped QPDG consists of those statements that can transmit a change from the source to the target . By building a chopped QPDG, the statements of the program which do not contribute to bring secure information flows from high input to low output are filtered. We therefore can provide a more effective analysis by focusing on the essential parts which introduces the unwellcome flows.

Theorem 2 (Dependences and information flow). *Given a QPDG $G = (N, E_x)$, if there is a backward path from an output variable y typed as L at target node to an input variable x typed as H at source node, there is a dependence relationship between them: $\Delta(y \mapsto x)$, and there is information flow from x to y: $x \rightsquigarrow y$.*

Proof. Follows from Definition 1 and Definition 3. If we view the reduced QPDG as a single box labelled by l_{RG}, it conveyed the dependence relations between x and y directly. Therefore the quantity of the information flow from x to y can be viewed as the content of the dependence introduced by l_{RG}. i.e., the flow quantity from x to y is computed by: $d(x \rightsquigarrow y) = q(\Delta_{l_{RG}}(y \mapsto x))$, where X, Y denotes the random variable of the program variables x, y, Z denotes the joint random variable of other program variables on which y depends.

Clearly, if the quantity of the dependence between them is 0, there is no information flow from high input to low output, i.e., the program satisfies *non-interference* and is secure. Specifically, the analysed program S_l is considered secure if $\forall x \in L$, there is no $y \in H$ such that $\Delta_l(x \mapsto y)$, i.e., there is no low level variable at the target node depending on high level variables at the source node. We therefore can present the security condition as follows.

Definition 4. *Security condition.* $\forall x \in L$, *there is no* $y \in H$ *such that* $\Delta(x \mapsto \delta) \ \wedge \ y \in \delta$.

6 Conclusions

We have introduced a definition of *Quantitative Program Dependence Graph* (QPDG), which can be used to capture the quantified dependencies of programs given the source node and target node of interest. We presented the semantics in order to build a QPDG, and suggested a method to provide a further abstraction on the QPDG given the target node and source node by doing backward slicing. Since such a graph can show the information flow on a data slice explicitly, as an application, it can represent well the interesting elementary changes of the information flow of a program. We believe that the QPDGs can be useful in a number of area in program analysis and software engineering in quantitative aspects, e.g., program verification, fault localisation, and program optimisation etc. For future work, we propose to explore an application of the reduced QPDG for localising faults in programs with quantitative reasoning. For instance, we propose to explore algorithms to calculate the degree of the suspicious of the statements in order to localise the possible faults regarding to different requirements. We also plan to extend our language and graphs to a richer version to allow procedures and capture concurrent behaviours.

Acknowledgements. Many thanks to Ian Hayes, Cliff Jones, Ken Pierce, Carl Gamble and the anonymous reviewers for discussions and comments.

References

1. Baah, G.K., Podgurski, A., Harrold, M.J.: The probabilistic program dependence graph and its application to fault diagnosis. IEEE Trans. Software Eng. 36(4), 528–545 (2010)

2. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: S & P (2009)
3. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical Measurement of Information Leakage. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 390–404. Springer, Heidelberg (2010)
4. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15, 321–371 (2007)
5. Clark, D., Hankin, C., Hunt, S.: Information flow for algol-like languages. Comput. Lang. 28(1), 3–28 (2002)
6. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
7. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: ACSAC, pp. 261–269 (2010)
8. Jackson, D., Rollins, E.J.: A new model of program dependences for reverse engineering. In: SIGSOFT FSE, pp. 2–10 (1994)
9. Jackson, D.: Aspect: Detecting bugs with abstract dependences. ACM Trans. Softw. Eng. Methodol. 4(2), 109–145 (1995)
10. Kuper, R.: Dependency-directed localization of software bugs (1989)
11. Malacaria, P.: Assessing security threats of looping constructs. In: POPL, pp. 225–235. ACM Press, Nice (2007)
12. Mateis, C., Stumptner, M., Wieland, D., Wotawa, F.: Model-based debugging of java programs. In: AADEBUG (2000)
13. McIver, A., Meinicke, L., Morgan, C.: Hidden-markov program algebra with iteration. CoRR abs/1102.0333 (2011)
14. Moriconi, M., Winkler, T.C.: Approximate reasoning about the semantic effects of program changes. IEEE Transactions on Software Engineering 16, 980–992 (1990)
15. Mu, C., Clark, D.: An interval-based abstraction for quantifying information flow. In: ENTCS, vol. 59, pp. 119–141. Elsevier (2009)
16. Mu, C., Clark, D.: Quantitative analysis of secure information flow via probabilistic semantics. In: ARES, pp. 49–57 (2009)
17. Ottenstein, K.J., Ottenstein, L.M.: The program dependence graph in a software development environment. In: Software Development Environments (SDE), pp. 177–184 (1984)
18. Rudin, W.: Real and Complex Analysis. McGraw-Hill (1966)
19. Shannon, C.E.: A mathematical theory of communication. SIGMOBILE Mob. Comput. Commun. Rev. 5(1), 3–55 (1948)
20. Weise, D., Crew, R.F., Ernst, M., Steensgaard, B.: Value dependence graphs: Representation without taxation. In: POPL, pp. 297–310 (1994)
21. Weiser, M.D.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, aAI8007856 (1979)
22. Wotawa, F., Soomro, S.: Fault Localization Based on Abstract Dependencies. In: Ali, M., Esposito, F. (eds.) IEA/AIE 2005. LNCS (LNAI), vol. 3533, pp. 357–359. Springer, Heidelberg (2005)

Quantitative Analysis of Information Flow Using Theorem Proving

Tarek Mhamdi, Osman Hasan, and Sofiène Tahar

ECE Department, Concordia University, Montreal, QC, Canada
{mhamdi,o_hasan,tahar}@ece.concordia.ca

Abstract. Quantitative analysis of information flow is widely used to measure how much information was leaked from the secret inputs to the outputs or public inputs of a program. We propose to conduct the quantitative analysis of information flow within the trusted kernel of a higher-order-logic theorem prover in order to overcome the inaccuracy limitations of traditional analysis techniques used in this domain. For this purpose, we present the formalization of the Kullback-Leibler divergence that can be used as a unified measure of information leakage. Furthermore, we propose two new measures of information leakage, namely the information leakage degree and the conditional information leakage degree. We also formalize the notion of anonymity-based single MIX and use the channel capacity as a measure of information leakage in the MIX. Finally, for illustration purposes, we show how our framework allowed us to find a counter-example for a theorem that was reported in the literature to describe the leakage properties of the anonymity-based single MIX.

1 Introduction

Quantitative information flow [19,17] allows to measure how much information about the high security inputs of a system can be leaked, accidentally or maliciously, by observing the systems outputs and possibly the low security inputs. Unlike non-interference analysis, which only determines whether a system is completely secure or not completely secure, quantitative information flow provides an information theoretic measure on how secure or insecure a system is. Quantitative information flow is extensively used for analyzing anonymity protocols and secure communications using various measures of information flow. Serjantov [18] and Diaz et al. [6] independently proposed to use the entropy to define the quality of anonymity and to compare different anonymity systems. Malacaria [12] defined the leakage of confidential information in a program as the conditional mutual information between its outputs and secret inputs, given the knowledge of its low security inputs. Deng [5] proposed relative entropy as a measure of the amount of information revealed to the attacker after observing the outcomes of the protocol, together with the a priori information. Chatzikokolakis [1] modeled anonymity protocols as noisy channels and used the channel capacity as a measure of the loss of anonymity.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 119–134, 2012.

Traditionally, paper-and-pencil based analysis or computer simulations have been used for quantitative analysis of information flow. Paper-and pencil analysis does not scale well to complex systems and is prone to human error. Computer simulation, on the other hand, lacks in accuracy due to numerical approximations. These analysis inaccuracies may result in compromising national security and finances given the safety and security-critical nature of systems where information flow analysis is usually used.

As an alternative approach, we propose a machine-assisted analysis of information flow by conducting the analysis within the trusted kernel of a higher-order-logic theorem prover [9]. Higher-order logic is a system of deduction with precise semantics and, due to its high expressiveness, can be used to describe any mathematical relationship. Interactive theorem proving is the field of computer science and mathematical logic concerned with computer-based formal proof tools that require human assistance. We argue that the high expressiveness of higher-order logic can be leveraged to formalize the commonly used information leakage measures by building upon the existing formalization of measure, integration and probability [13], and information theories [14]. This foundational formalization can hence be used to formally reason about quantitative properties of information flow analysis within the sound core of a theorem prover and thus guarantee accuracy of the analysis.

In particular, this paper presents an extension of existing theories of measure, Lebesgue integration and probability [13] to cater for measures involving multiple random variables. Building upon this formalization, we present a higher-order-logic formalization of the Kullback-Leibler (KL) divergence [4] from which we can derive the formalization of most of the information leakage measures presented in the literature so far. Furthermore, we propose two novel measures of information leakage termed as degrees of information leakage. We will show that they are somehow related to the existing measures but have the advantage that they not only quantify the information leakage but also describe the quality of leakage by normalizing the measure by the maximum leakage that the system allows under extreme situations. The formalization reported in this paper has been done using the HOL4 [10] theorem prover. The prime motivation behind this choice is the availability of the measure, probability and integration theories [13,14].

We illustrate the usefulness of the framework for formal analysis of quantitative information flow by tackling an anonymity-based single MIX application [20]. We provide a higher-order-logic formalization of the single MIX as well as the channel capacity which we use as a measure of information leakage within the MIX. We then formally verify that a sender using the MIX as a covert channel, can transmit information through the MIX at maximum capacity without having to communicate with all the receivers. This result allowed us to identify a flaw in the paper-and-pencil based analysis of a similar problem [20] which clearly indicates the usefulness of the proposed technique.

The rest of the paper is organized as follows: In Section 2 we briefly present the proposed extensions to the formalization of measure, integration and probability theories [13]. In Section 3, we describe our formalization of KL divergence

and how we use it to formalize various measures of information leakage as well as prove their properties in HOL. We introduce novel information leakage degrees in Section 4. In Section 5, we present an analysis of an anonymity-based single MIX. We discuss related work in Section 6 and conclude the paper in Section 7.

2 Measure, Integration and Probabilities

In this section, we extend the formalization of measure theory [13] and Lebesgue integration [13] as well as probability theory [14] to support products of measure spaces and joint distributions of two or more random variables. Both are necessary to define measures over multiple random variables.

Products of Measure Spaces. Let $m_1 = (X_1, \mathcal{S}_1, \mu_1)$ and $m_2 = (X_2, \mathcal{S}_2, \mu_2)$ be two measure spaces. The product of m_1 and m_2 is defined to be the measure space $(X_1 \times X_2, \mathcal{S}, \mu)$, where $\mathcal{S}$ is the sigma algebra on $X_1 \times X_2$ generated by subsets of the form $A_1 \times A_2$ where $A_1 \in \mathcal{S}_1$, and $A_2 \in \mathcal{S}_2$. The measure μ is defined for σ-finite measure spaces as

$$\mu(A) = \int_{X_1} \mu_2(\{y \in X_2 | (x, y) \in A\})\, d\mu_1$$

and $\mathcal{S}$ is defined using the `sigma` operator which returns the smallest sigma algebra containing a set of subsets, i.e., the product subsets in this case.

Let $g(s_1)$ be the function $s_2 \to (s_1, s_2)$ and `PREIMAGE` denote the HOL function for inverse image, then the product measure is formalized as

```
⊢ prod_measure m1 m2 =
    (λa. integral m1 (λs1. measure m2 (PREIMAGE g(s1) a)))
```

We verified in HOL that the product measure can be reduced to $\mu(a_1 \times a_2) = \mu_1(a_1) \times \mu_2(a_2)$ for finite measure spaces.

```
⊢ prod_measure m1 m2 (a1 × a2) = measure m1 a1 × measure m2 a2
```

We use the above definitions to define products of more than two measure spaces as follows. $X_1 \times X_2 \times X_3 = X_1 \times (X_2 \times X_3)$ and $\mu_1 \times \mu_2 \times \mu_3$ is defined as $\mu_1 \times (\mu_2 \times \mu_3)$. We also define the notion of absolutely continuous measures where μ_1 is said to be absolutely continuous w.r.t μ_2 iff for every measurable set A, $\mu_2(A) = 0$ implies $\mu_1(A) = 0$. Further details about this formalization can be found in [15].

Joint Distribution. The joint distribution of two random variables defined on the same probability space is defined as,

$$p_{XY}(a) = p(\{(X, Y) \in a\})$$

```
⊢ joint_distribution p X Y =
        (λa. prob p (PREIMAGE (λx. (X x,Y x)) a ∩ Ω))
```

Here the intersection with the sample space Ω is required because HOL functions are total and should be defined on all variables of the specific type instead of only on Ω. The joint distribution of any number of variables can be defined in a similar way. We formally verified a number of joint distribution properties in HOL [15] and some of the useful ones are given below:

```
⊢ 0 ≤ joint_distribution p X Y a
⊢ joint_distribution p X Y = joint_distribution p Y X
⊢ joint_distribution p X Y (a × b)  ≤ distribution p X a
⊢ joint_distribution p X Y (a × b)  ≤ distribution p Y b
```

We also verified that the joint distribution is absolutely continuous w.r.t to the product of marginal distributions and the following useful properties in HOL.

$$p_X(a) = \sum_{y \in Y(\Omega)} p_{XY}(a \times \{y\})$$

$$p_Y(b) = \sum_{x \in X(\Omega)} p_{XY}(\{x\} \times b)$$

The formalization of joint distributions and products of measures spaces, presented in the next section, play a vital role in formalizing information-theoretic measures with multiple random variables.

3 Measures of Information Leakage

In this section, we first provide a formalization of the Radon-Nikodym derivative [8] which is then used to define the KL divergence. Based on the latter, we define most of the commonly used measures of information leakage. We start by providing general definitions which are valid for both discrete and continuous cases and then prove the corresponding reduced expressions where the measures considered are absolutely continuous over finite spaces.

3.1 Radon-Nikodym Derivative

The Radon-Nikodym derivative of a measure ν with respect to the measure μ is defined as a non-negative measurable function f, satisfying the following formula, for any measurable set A.

$$\int_A f \, d\mu = \nu(A)$$

We formalize the Radon-Nikodym derivative in HOL as

```
⊢ RN_deriv m v =
   @f. f IN measurable (X, S) Borel ∧
   ∀x ∈ X, 0 ≤ f x ∧
   ∀a ∈ S, integral m (λx. f x * indicator_fn a x) = v a
```

where @ denotes the Hilbert-choice operator in HOL. The existence of the Radon-Nikodym derivative is guaranteed for absolutely continuous measures by the Radon-Nikodym theorem.

Theorem 1. *If ν is absolutely continuous with respect to μ, then there exists a non-negative measurable function f such that for any measurable set A,*

$$\int_A f \, d\mu = \nu(A)$$

We proved the Radon-Nikodym theorem in HOL for finite measures which can be easily generalized to σ-finite measures.

```
⊢ ∀m v s st.
    measure_space (s,st,m) ∧ measure_space (s,st,v) ∧
    measure_absolutely_continuous (s,st,m) (s,st,v) ∧
    v s ≠ ∞ ∧ m s ≠ ∞ ⇒
       ∃f. f ∈ measurable (s,st) Borel ∧
       ∀x ∈ s, 0 ≤ f x < ∞ ∧
       ∀a ∈ st,
          integral m (λx. f x * indicator_fn a x) = v a
```

The formal reasoning about the above theorem is primarily based on the Lebesgue monotone convergence and the following lemma which, to the best of our knowledge, has not been referred to in paper-and-pencil based mathematical texts before.

Lemma 1. *If P is a non-empty set of extended-real valued functions closed under the max operator, g is monotone over P and $g(P)$ is upper bounded, then there exists a monotonically increasing sequence $f(n)$ of functions, elements of P, such that*

$$\sup_{n \in \mathbb{N}} g(f(n)) = \sup_{f \in P} g(f)$$

Finally, we formally verified various properties of the Radon-Nikodym derivative. For instance, we prove that for absolutely continuous measures defined over a finite space, the derivative reduces to

```
⊢ ∀x ∈ s, u{x} ≠ 0 ⇒ RN_deriv u v x = v{x} / u{x}
```

The following properties play a vital role in formally reasoning about the Radon-Nikodym derivative and have also been formally verified.

```
⊢ ∀x ∈ s, 0 ≤ RN_deriv m v x < ∞
⊢ RN_deriv ∈ measurable (s,st) Borel
⊢ ∀a ∈ st, integral m (λx. RN_deriv m v x * indicator_fn a x) = v a
```

3.2 Kullback-Leibler Divergence

The Kullback-Leibler (KL) divergence [4] $D_{KL}(\mu||\nu)$ is a measure of the distance between two distributions μ and ν. It can be used to define most information-theoretic measures such as the mutual information and entropy and can, hence, be used to provide a unified framework to formalize most information leakage measures. It is because of this reason that we propose to formalize the KL divergence in this paper as it will facilitate formal reasoning about a wide variety of information flow related properties. The KL divergence is defined as

$$D_{KL}(\mu||\nu) = -\int_X log\frac{d\nu}{d\mu}\, d\mu$$

where $\frac{d\nu}{d\mu}$ is the Radon-Nikodym derivative of ν with respect to μ. The KL divergence is formalized in HOL as

```
⊢ KL_divergence b m v = -integral m (λx. logr b((RN_deriv m v)x))
```

where b is the base of the logarithm. D_{KL} is measured in *bits* when $b = 2$. We formally verify various properties of the KL divergence. For instance, we prove that for absolutely continuous measures over a finite space, it reduces to

$$D_{KL}(\mu||\nu) = \sum_{x \in s} \mu\{x\} \log \frac{\mu\{x\}}{\nu\{x\}}$$

```
⊢ KL_divergence b u v = SIGMA (λx. u{x} logr b (u{x} / v{x})) s
```

We also prove the following properties

```
⊢ KL_divergence b u u = 0
⊢ 1 ≤ b  ⇒  0 ≤ KL_divergence b u v
```

The non-negativity of the KL divergence for absolutely continuous probability measures over finite spaces is extensively used to prove the properties of information theory measures like the mutual information and entropy. To prove this result, we use the Jensen's inequality and the concavity of the logarithm function.

We show in the subsequent sections how we use the KL divergence to formalize the mutual information, Shannon entropy, conditional entropy and the conditional mutual information, which are some of the most commonly used measures of information leakage.

3.3 Mutual Information and Entropy

The mutual information has been proposed as a measure of information leakage [20] from the secure inputs S of a program to its public outputs O as it represents the mutual dependence between the two random variables S and O. The mutual information is defined as the KL divergence between the joint distribution and the product of marginal distributions. The following is a formalization of the mutual information in HOL.

```
⊢ I(X;Y) = KL_divergence b (joint_distribution p X Y)
                           prod_measure (distribution p X)
                                        (distribution p Y)
```

We prove various properties of the mutual information in HOL, such as the non-negativity, symmetry and reduced expression for finite spaces, using the result that the joint distribution is absolutely continuous w.r.t the product of marginal distributions.

```
⊢ 0 ≤ I(X;Y)
⊢ I(X;Y) = I(Y;X)
⊢ I(X;Y) = 0 ⇔ X and Y independent
⊢ I(X;Y) = SIGMA (λ(x,y). p{(x,y)} logr b (p{(x,y)}/p{x}p{y})) s
```

The Shannon entropy `H(X)` was one of the first measures to be proposed to analyze anonymity protocols and secure communications [18,6] as it intuitively measures the uncertainty of a random variable `X`. It can be defined as the expectation of p_X or simply as $I(X;X)$.

```
⊢ H(X) = I(X;X)
```

We prove that it can also be expressed in terms of the KL divergence between p_X and the uniform distribution p_X^u, where N is the size of the alphabet of X.

```
⊢ H(X) = log(N) - KL_divergence b (distribution p X)
                                  (uniform_dist p X)
```

The cross entropy $H(X,Y)$ is the entropy of the random variable (X,Y) and hence there is no need for a separate formalization of the cross entropy.
The conditional entropy is defined in terms of the KL divergence as follows:

```
⊢ H(X|Y) = log(N) - KL_divergence b (joint_distribution p X Y)
                                    prod_measure (uniform_dist p X)
                                                 (distribution p Y)
```

The entropy properties that we prove in HOL include:

```
⊢ 0 ≤ H(X) ≤ log(N)
⊢ max(H(X),H(Y)) ≤ H(X,Y) ≤ H(X) + H(Y)
⊢ H(X|Y) = H(X,Y) - H(Y)
⊢ 0 ≤ H(X|Y) ≤ H(X)
⊢ I(X;Y) = H(X) + H(Y) - H(X,Y)
⊢ I(X;Y)  ≤ min(H(X),H(Y))
⊢ H(X)   = -SIGMA (λx. p{x} logr b (p{x})) s
⊢ H(X|Y) = -SIGMA (λ(x,y). p{(x,y)} logr b (p{(x,y)}/p{y})) s
```

3.4 Conditional Mutual Information

The conditional mutual information $I(X;Y|Z)$ allows one to measure how much information about the secret inputs X is leaked to the attacker by observing the outputs Y of a program given knowledge of the low security inputs Z. This property was used by Malacaria [12] to introduce the conditional mutual information as a measure of information flow for a program with high security inputs and low security inputs and outputs. The conditional mutual information is defined as the KL divergence between the joint distribution p_{XYZ} and the product measure $p_{X|Z}p_{Y|Z}p_Z$. The HOL formalization is as follows.

```
⊢ conditional_mutual_information b p X Y Z =
   KL_divergence b (joint_distribution p X Y Z)
                   (prod_measure (conditional_distribution p X Z)
                                 (conditional_distribution p Y Z)
                                 (distribution p Y))
```

We formally verify the following reduced form of the conditional mutual information for finite spaces by first proving that p_{XYZ} is absolutely continuous w.r.t $p_{X|Z}p_{Y|Z}p_Z$ and then apply the reduced form of the KL divergence.

$$I(X;Y|Z) = \sum_{(x,y,z)\in\mathcal{X}\times\mathcal{Y}\times\mathcal{Z}} p(x,y,z)\log\frac{p(x,y,z)}{p(x|z)p(y|z)p(z)}$$

When the two random variables X and Y are independent given Z, the conditional mutual information $I(X;Y|Z) = 0$. In fact, in this case,
$\forall x, y, z.\ p(x,y,z) = p(x,y|z)p(z) = p(x|z)p(y|z)p(z)$.

```
⊢ indep_rv_cond p X Y Z ⇒ I(X;Y|Z) = 0
```

We also prove a few other important results regarding the conditional mutual information which will be useful later in our work.

```
⊢ 0 ≤ I(X;Y|Z)
⊢ I(X;Y|Z) = H(X|Z) - H(X|Y,Z)
⊢ I(X;Y|Z) = I(X;(Y,Z)) - I(X;Z)
⊢ I(X;Y|Z) ≤ H(X|Z)
```

So far, we have provided a higher-order-logic formalization of the KL divergence which we used to define various measures of quantitative information flow. This framework, along with the formalization of measure and probability theories, allows us to conduct many analyses of quantitative information flow using a theorem prover and hence guaranteeing the soundness of the analysis.

4 Degrees of Information Leakage

We introduce two new measures of information leakage that can be used to describe the anonymity properties of security systems and protocols, namely the information leakage degree and the conditional information leakage degree.

4.1 Information Leakage Degree

We define the information leakage degree between random variables X and Y representing the secret inputs and public outputs of a program, respectively, as

$$D = \frac{H(X|Y)}{H(X)}$$

```
⊢ information_leakage_degree p X Y =
        conditional_entropy p X Y / entropy p X
```

To better understand the intuition behind this definition, let us consider the two extreme cases of a completely secure program and a completely insecure program. Complete security, intuitively, happens when the knowledge of the public output Y of a program does not affect the uncertainty about the secret input X. This is equivalent to the requirement that X is independent of Y. In this case $H(X|Y) = H(X)$ and the information leakage degree is equal to 1. On the other hand, when the output of the program completely identifies its secret input, the entropy $H(X|Y)$ is equal to 0 and hence the information leakage degree is equal to 0 in this case of perfect identification. For situations between the two extremes, the information leakage degree lies in the interval $(0, 1)$. We formally verified this result as the following theorem in HOL which provides the bounds of the degree of information leakage.

```
⊢ 0 ≤ information_leakage_degree p X Y ≤ 1
```

Using the properties of the mutual information we prove that the information leakage degree is also equal to

$$D = 1 - \frac{I(X;Y)}{H(X)}$$

This result illustrates the significance of the information leakage degree definition since the mutual information measures how much information an adversary can learn about the input X after observing the output Y. This also allows to compare our definition to the anonymity degree proposed in [20] as

$$D' = 1 - \frac{I(X;Y)}{logN}$$

where N is the size of the alphabet of X. Our definition is more general. In fact, when X is uniformly distributed, the two measures coincide $D = D'$. However in the general case we believe that our definition is more accurate since in the perfect identification scenario, for instance, D is always equal to 1 regardless of the input distribution. On the other hand, D' is equal to 1 only in the special case of a uniform distribution. In [20] the authors considered using $H(X)$ as a normalization factor instead of $logN$ but opted for the latter arguing that the input distribution is already accounted for in the mutual information. But as stated previously, with the definition of D', the proof for perfect identification is only valid for uniformly distributed inputs.

4.2 Conditional Information Leakage Degree

We propose another variation of information leakage degree that is more general and can cover a wider range of scenarios. First, consider a program which has a set of high security inputs S, a set of low security inputs L and a set of public outputs O. The adversary wants to learn about the high inputs S by observing the outputs O given the knowledge of the low inputs L. To capture this added information to the adversary (low inputs), we propose the following definition, which we call the conditional information leakage degree.

$$D_c = \frac{H(S|(O,L))}{H(S|L)}$$

This can be formalized in HOL as

```
⊢ conditional_information_leakage_degree p S L O =
    conditional_entropy p S (O,L) / conditional_entropy p S L
```

Just like the previous case, consider the two extremes of perfect security and perfect identification. When the outputs and the secret inputs are independent, given L, the conditional entropy $H(S|(O,L))$ is equal to $H(S|L)$ which results in a conditional leakage degree equal to 1 for perfect security. However, if the public inputs and outputs completely identify the secret inputs, then $H(S|(O,L))$ is equal to 0 and so is the conditional leakage degree in the case of perfect identification. As in the case of leakage degree, we are also able to prove that the conditional information leakage degree lies in the interval $[0,1]$.

```
⊢ 0 ≤ conditional_information_leakage_degree p X Y Z ≤ 1
```

We also prove that the conditional information leakage degree can be written in terms of the conditional mutual information and the conditional entropy.

$$D = 1 - \frac{I(S;O|L)}{H(S|L)}$$

This shows that this definition is clearly a generalization of the information leakage degree for the case of programs with additional low security inputs. We provide more intuition to interpret this definition by proving the data processing inequality (DPI) [4].

Definition 1. *Random variables X, Y, Z are said to form a Markov chain is that order (denoted by $X \to Y \to Z$) if the conditional distribution of Z depends only on Y and is conditionally independent of X. Specifically, X, Y and Z form a Markov chain $X \to Y \to Z$ if the joint probability mass function can be written as $p(x,y,z) = p(x)p(y|x)p(z|y)$.*

We formalize this in HOL as follows.

```
⊢ markov_chain p X Y Z =
   ∀ x y z. joint_distribution p X Y Z {(x,y,z)} =
                distribution p X {x} *
                conditional_distribution p Y X {(y,x)} *
                conditional_distribution p Z Y {(z,y)}
```

We prove that $X \to Y \to Z$ is equivalent to the statement that X and Z are conditionally independent given Y. In fact, $p(x)p(y|x)p(z|y) = p(x,y)p(z|y) = p(x|y)p(z|y)p(y)$. This in turn is equivalent to $I(X;Z|Y) = 0$. This result will allow us to prove the DPI theorem.

Theorem 2. *(DPI) if* $X \to Y \to Z$ *then* $I(X;Z) \leq I(X;Y)$

```
⊢ markov_chain p X Y Z ⇒
    mutual_information b p X Z ≤ mutual_information b p X Y
```

We prove the DPI theorem using the properties of the mutual information. In fact, as shown previously, $I(X;(Y,Z)) = I(X;Z) + I(X;Y|Z)$. By symmetry of the mutual information, we also have $I(X;(Y,Z)) = I(X;Y) + I(X;Z|Y) = I(X;Y)$. The last equality results from the fact that $I(X;Z|Y) = 0$ for a Markov Chain. Using the non-negativity of the conditional mutual information, which we proved previously, it is straightforward to conclude that $I(X;Z) \leq I(X;Y)$.

The data processing inequality is an important result in information theory that is used, for instance, in statistics to define the notion of sufficient statistic. We make use of the DPI to interpret the conditional information leakage degree. For a system with high security inputs S, low security inputs L and outputs O, if the outputs depend only on the low inputs, i.e., $p(O|S,L) = p(O|L)$ then $S \to L \to O$ and S and O are conditionally independent given L. This is the perfect security scenario, for which $D_c = 1$. Using the DPI, we conclude that $I(S;O) \leq I(S;L)$. This means that when the conditional mutual information leakage is equal to 1, no clever manipulation of the low inputs, by the attacker, deterministic or random, can increase the information that L contains about S $(I(S;L))$.

We have presented so far our higher-order-logic formalization of measures of information flow building upon the extension of measure, Lebesgue integration and probability formalization in HOL. Overall the HOL definitions and proof scripts of the above formalization required around 15,000 lines of code [15]. These results can now be readily used to reason about information flow analysis of real-world protocols and programs.

5 Application

In this section, we use our formalization to reason about an anonymity-based single MIX, designed to hide the communication links between a set of senders and a set of receivers. We model a single MIX as a communication node connecting m senders $(s_1, \ldots, s_m)$ to n receivers $(r_1, \ldots, r_n)$. The single MIX is determined by its inputs (senders), outputs (receivers) and the transition probabilities. We can also add clauses in the specification to capture additional information about the MIX like structural symmetry. The following is the formalization of the single MIX given in Figure 1.

```
⊢ MIX_channel s m X Y =
   (IMAGE X s = {0;1}) ∧ (IMAGE Y s = {0;1;2;3})  ∧
   (conditional_distribution (s,POW s,m) Y X {0} {0} = 1/2) ∧
   (conditional_distribution (s,POW s,m) Y X {1} {0} = 1/2) ∧
   (conditional_distribution (s,POW s,m) Y X {2} {1} = 1)
```

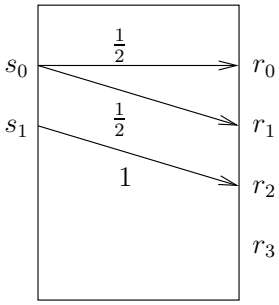


Fig. 1. Single MIX

Zhu and Bettati [20] used the single MIX to model an anonymity-based covert-channel where a sender is trying to covertly send messages through the MIX. They used the channel capacity as a measure of the maximum information that can be leaked through the MIX and can be used as a measure of the quality of anonymity of the network. A communication between a sender s_i and a receiver r_j is denoted by $[s_i, r_j]$. The term $p([s_u, r_v]_s|[s_i, r_j]_a)$ represents the probability that the communication $[s_u, r_v]$ is suspected given that $[s_i, r_j]$ is actually taking place. This model describes attacks on sender-receiver anonymity. The input symbols of the covert-channel are the actual sender-receiver pairs $[s, r]_a$ and the output symbols are the suspected pairs $[s, r]_s$. In this case, $p([s, r]_s|[s, r]_a)$ represents the result of the anonymity attack. We consider the case where an attacker can establish a covert-channel by having 1 sender s_1 communicate with any combination of j receivers. The same reasoning can be applied to multiple senders. The authors claim the following result [20]

Lemma 2. *For a single sender s_1 on a single mix, the maximum covert-channel capacity is achieved when s_1 can communicate to all receivers.*

We initially tried to formally verify this result, using the foundational results presented in the previous two sections of this paper, but we found a counter-example for an assumption upon which the paper-and-pencil proof of Lemma 2 is based [20]. The erroneous assumption states that the maximum of the mutual information is achieved when all input symbols have non-zero probabilities regardless of the transition probabilities (the results of the anonymity attack). We are able to prove in HOL that it is not necessary for the sender s_1 to communicate with all receivers to achieve capacity.

First, we provide a higher-logic-formalization of the channel capacity which is defined as the maximum, over all input distributions, of the mutual information between the input and the output of the channel. We formalize it in HOL using the Hilbert-choice operator; i.e., if it exists, the capacity is some c such that $c = I_m(X;Y)$ for some probability distribution m and for any input distribution p, $I_p(X;Y) \leq c$.

```
⊢ capacity s X Y = @c.
      ∃m. c = mutual_information (s, POW s, m) X Y  ∧
      ∀m. mutual_information (s, POW s, m) X Y ≤ c
```

Next, consider the covert-channel depicted in Figure 2. To simplify the notation, let $x_i = [s_1, r_i]_a$ and $y_i = [s_1, r_i]_s$. This covert-channel is formalized in HOL as

```
⊢ MIX_channel_1 s m X Y =
   (IMAGE X s = {0;1;2}) ∧ (IMAGE Y s = {0;1;2})  ∧
   (distribution(s,POW s,m) X{0} = distribution(s,POW s,m) X{2}) ∧
   (conditional_distribution (s,POW s,m) Y X {0} {0} = 1) ∧
   (conditional_distribution (s,POW s,m) Y X {0} {1} = 1 / 2) ∧
   (conditional_distribution (s,POW s,m) Y X {0} {2} = 0) ∧
   (conditional_distribution (s,POW s,m) Y X {1} {0} = 0) ∧
   (conditional_distribution (s,POW s,m) Y X {1} {1} = 0) ∧
   (conditional_distribution (s,POW s,m) Y X {1} {2} = 0) ∧
   (conditional_distribution (s,POW s,m) Y X {2} {0} = 0) ∧
   (conditional_distribution (s,POW s,m) Y X {2} {1} = 1 / 2) ∧
   (conditional_distribution (s,POW s,m) Y X {2} {2} = 1)
```

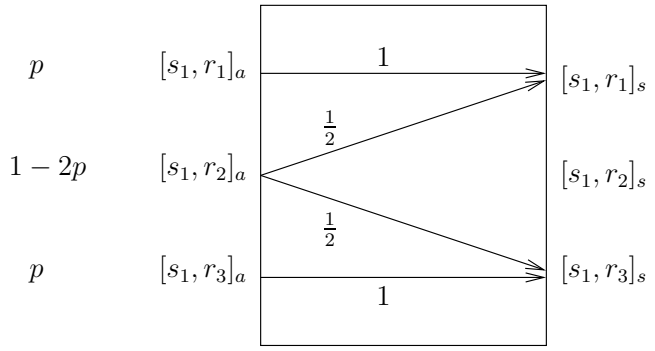


Fig. 2. Single MIX example

We prove that its mutual information is equal to $2p$.

```
⊢ ∀X Y s. MIX_channel_1 s m X Y ⇒
       mutual_information 2 (s, POW s, m) X Y =
          2 * distribution (s, POW s, m) X {0}
```

We also prove that the capacity is equal to 1 and corresponds to $p = \frac{1}{2}$. This means that the input distribution that achieves the channel capacity is $[p\{x_0\} = \frac{1}{2}, p\{x_1\} = 0, p\{x_2\} = \frac{1}{2}]$. Hence, we prove that the sender s_1 does not need to communicate with the receiver r_2 and still achieve maximum capacity, contradicting Lemma 2. Notice that with $p = \frac{1}{2}$, $I(X;Y) = H(X) = 1$ which implies that the degree of information leakage $D = 0$. So for this covert-channel, the maximum capacity corresponds to perfect identification.

Unlike the paper-and-pencil based analysis, a machine-assisted analysis of quantitative information flow using theorem proving guarantees the accuracy of the results. In fact, the soundness of theorem proving inherently ensures that only valid formulas are provable. The requirement that every single step of the proof needs to be derived from axioms or previous theorems using inference rules, allows us to find missing assumptions and even sometimes wrong statements as was the case in the single MIX application. We were able to detect the problem with the reasoning and confirm the result using our formalization in HOL.

6 Related Work

The underlying theories over which we built this work are mainly from [13] and [14]. In [13], we provided a formalization of the measure theory and Lebesgue integration in HOL and proved some classical probability results like the Weak Law of Large Numbers. In [14], we formalized extended reals and based on them provided a more extensive formalization of measure and Lebesgue integration. We also formalized the Shannon entropy and Relative entropy and proved the Asymptotic Equipartition Property. In the current paper, we enrich the underlying theories by adding, for instance, products of measure spaces and joint distributions. The main difference, however, is that in this paper we propose new measures of information leakage and formalize various other measures like mutual information and conditional mutual information based on a unified definition of the KL divergence. We also formalize the channel capacity and the notion of single MIX and use the framework for an illustrative example.

Coble [3] formalized some information theory in higher-order logic and used Malacaria's measure of information leakage, i.e., the conditional mutual information [12], to formally analyse the anonymity properties of the Dining Cryptographers protocol. Our formalization of information theory is an extended version of Coble's formalization, i.e., it supports Borel spaces and extended real numbers which allowed us to prove the Radon Nikodym theorem. Coble's formalization of information theory does not offer these capabilities and thus cannot be used to formally verify the Radon Nikodym theorem.

Zhu and Bettati [20] proposed the notion of degree of anonymity which is close to our definition of information leakage degree but we showed that our definition is more general and the two are equal in the case of uniform distribution. Besides, we proposed the conditional information leakage degree, suitable for programs with low security inputs and proved the data processing inequality to give more insight into the intuition behind this new definition. Moreover, our work is based on higher-order-logic theorem proving, which is arguably more sound than the

paper-and-pencil based analysis of Zhu and Bettati. In fact, with our analysis we were able to detect the aforementioned problem with the analysis in [20] and provide a counter-example using theorem proving.

Chatzikokolakis [1] modeled anonymity protocols as noisy channels and used the channel capacity as a measure of the loss of anonymity. In the case where some leakage is intended by design, like in an election protocol, they introduced the notion of conditional capacity which is related to the conditional mutual information. They used the PRISM model checker [11] to assist in computing the transition probabilities and capacity of two protocols, namely the Dining cryptographers and the Crowds protocol. This probabilistic model checking based analysis technique inherits the state-space explosion limitation of model checking. Similarly, it cannot be used to verify universally quantified generic mathematical relationships like we have been able to verify in the reported work.

7 Conclusions

In this paper, we conducted the quantitative analysis of information flow within the sound core of higher-order-logic theorem prover. For this purpose, we provided a formalization of the Kullback-Liebler divergence in the HOL4 theorem prover and used it to formalize various measures of information leakage that have been proposed in the literature such as the entropy, mutual information and conditional mutual information. We proposed two novel measures of information leakage which we called information leakage degree and gave some insight into the intuition behind the definitions.

We also provided a higher-order-logic formalization of channel capacity and the single MIX and used our framework in a small example to show the usefulness of using a theorem prover in this context. In fact, we were able to come up with a counter-example to a result that appeared in [20] related to the single MIX and proved in HOL that the senders need not communicate with all receivers to achieve channel capacity. Our results have been confirmed by Prof. Gallager from MIT, a well-known name in Information Theory and the author of the book Information Theory and Reliable Communication [7]. Catching this significant problem in the paper-and-pencil proofs clearly indicates the usefulness of using higher-order-logic theorem proving for conducting information flow analysis.

Our future plans include using this framework and new measures of information leakage to study the security properties of various protocols in HOL like the Dining Cryptographers [2] and Crowds protocols [16].

References

1. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity Protocols as Noisy Channels. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 281–300. Springer, Heidelberg (2007)
2. Chaum, D.: The Dining Cryptographers Problem: Unconditional Sender and Recipient Untraceability. Journal of Cryptology 1(1), 65–75 (1988)

3. Coble, A.R.: Formalized Information-Theoretic Proofs of Privacy Using the HOL4 Theorem-Prover. In: Borisov, N., Goldberg, I. (eds.) PETS 2008. LNCS, vol. 5134, pp. 77–98. Springer, Heidelberg (2008)
4. Cover, T.M., Thomas, J.A.: Elements of Information Theory. Wiley-Interscience (1991)
5. Deng, Y., Pang, J., Wu, P.: Measuring Anonymity with Relative Entropy. In: Dimitrakos, T., Martinelli, F., Ryan, P.Y.A., Schneider, S. (eds.) FAST 2006. LNCS, vol. 4691, pp. 65–79. Springer, Heidelberg (2007)
6. Díaz, C., Seys, S., Claessens, J., Preneel, B.: Towards Measuring Anonymity. In: Dingledine, R., Syverson, P.F. (eds.) PET 2002. LNCS, vol. 2482, pp. 54–68. Springer, Heidelberg (2003)
7. Gallager, R.G.: Information Theory and Reliable Communication. John Wiley & Sons, Inc. (1968)
8. Goldberg, R.R.: Methods of Real Analysis. Wiley (1976)
9. Gordon, M.J.C.: Mechanizing Programming Logics in Higher-Order Logic. In: Current Trends in Hardware Verification and Automated Theorem Proving, pp. 387–439. Springer (1989)
10. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: A Theorem Proving Environment for Higher-Order Logic. Cambridge University Press (1993)
11. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative Analysis with the Probabilistic Model Checker PRISM. Electronic Notes in Theoretical Computer Science 153(2), 5–31 (2005)
12. Malacaria, P.: Assessing Security Threats of Looping Constructs. SIGPLAN Notes 42(1), 225–235 (2007)
13. Mhamdi, T., Hasan, O., Tahar, S.: On the Formalization of the Lebesgue Integration Theory in HOL. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 387–402. Springer, Heidelberg (2010)
14. Mhamdi, T., Hasan, O., Tahar, S.: Formalization of Entropy Measures in HOL. In: van Eekelen, M., Geuvers, H., Schmaltz, J., Wiedijk, F. (eds.) ITP 2011. LNCS, vol. 6898, pp. 233–248. Springer, Heidelberg (2011)
15. Mhamdi, T., Hasan, O., Tahar, S.: Quantitative Information Flow Analysis in HOL (2012), http://hvg.ece.concordia.ca/code/hol/information-flow/
16. Reiter, M.K., Rubin, A.D.: Crowds: Anonymity for Web Transactions. ACM Transactions on Information and System Security 1(1), 66–92 (1998)
17. Sabelfeld, A., Myers, A.C.: Language-Based Information-Flow Security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
18. Serjantov, A., Danezis, G.: Towards an Information Theoretic Metric for Anonymity. In: Dingledine, R., Syverson, P.F. (eds.) PET 2002. LNCS, vol. 2482, pp. 41–53. Springer, Heidelberg (2003)
19. Smith, G.: On the Foundations of Quantitative Information Flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)
20. Zhu, Y., Bettati, R.: Information Leakage as a Model for Quality of Anonymity Networks. IEEE Transactions on Parallel and Distributed Systems 20(4), 540–552 (2009)

Modeling and Verification of Probabilistic Actor Systems Using pRebeca

Mahsa Varshosaz and Ramtin Khosravi

School of Electrical and Computer Engineering,
College of Engineering, University of Tehran, Tehran, Iran
{m.varshosaz,rkhosravi}@ece.ut.ac.ir

Abstract. Quantitative verification has gained an increasing attention as a promising approach for analysis of systems in various domains, especially for distributed systems, where the uncertainties of the environment cause the system to exhibit probabilistic and nondeterministic behavior. In this paper, we introduce pRebeca, an extension to the high-level actor-based modeling language Rebeca, that is used to model distributed and reactive systems with probabilistic and nondeterministic nature. We propose a simple syntax suitable for describing different aspects of a probabilistic system behavior and provide a formal semantics based on Markov decision processes. To model check a pRebeca model, it is converted to a Markov decision process and verified using the PRISM model checker against PCTL properties. Using a couple of examples, we show how a probabilistic system can be expressed in pRebeca in a simple way, while taking advantage of the PRISM model checker features.

Keywords: probabilistic model checking, actor model, pRebeca, Rebeca.

1 Introduction

As a broad range of computer systems exhibit probabilistic and nondeterministic behavior, there has been an increasing interest in employing and developing quantitative verification techniques to analyze and evaluate various properties of such systems. Distributed systems have been among motivating domains gaining much attention for application of quantitative techniques. The widespread and rapid growth of distributed systems such as sensor networks, web-based systems, etc., make it worth to analyze various quantitative and qualitative properties about them. As a starting point of analysis, modeling of such systems could take some effort due to structural and behavioral complexities. Thus, using a computational model compatible with the domain can effectively reduce this effort by expressing the concepts and entities of the domain directly. A modeling language based on the proper computational model, avoids putting extra effort to model the basic computations by providing proper primitives and level of abstraction, making the model more readable and maintainable. To this end, we introduce pRebeca which is an extension to the Rebeca (Reactive Objects

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 135–150, 2012.

Language) [1] language. It is a high-level object-based modeling language based on the actor model, suitable for describing asynchronous distributed systems with probabilistic behavior.

The actor model is one of the pioneering computing models for concurrent and asynchronous distributed systems. An actor model consists of a number of actors, which are the universal primitives in this model, communicating via asynchronous message passing. Since the introduction of actor as an agent-based language by Hewitt [2] and its development by Agha [3, 4] to a concurrent object based language, various interpretations and extensions of this model have been developed and are widely used in theory and practice [5–7]. Rebeca is an operational interpretation of actor model in the form of a high-level modeling language with simple Java-like syntax, formal semantics, and model checking tools which makes it proper and easy to use especially for asynchronous distributed systems. In a Rebeca model, rebecs (reactive objects) are actors, communicating via asynchronous message passing. Each rebec can perform computation on its local variables, send messages or create new rebecs in response to a received message.

In spite of widespread use of the actor model in both industry and academia, there has been little work done on modeling and verification of actor-based systems with probabilistic features. To the best of our knowledge, pRebeca is the first actor based modeling language with probabilistic modeling features. The only related work is PMAUDE [8] which is a specification language not based on the actor model but implementing actors with distribution probabilities on time of message passing and computation, without nondeterminism [8]. We extended Rebeca to make it capable of modeling actor-based systems with probabilistic and nondeterministic behavior. The simple and easy to learn syntax of pRebeca, makes it a suitable modeling language in practice. In a nutshell, in this work we present:

- Support for modeling probabilistic actor systems, via language constructs for:
  - Probabilistic choice between alternative behavior
  - Unreliable message passing between actors
- pRebeca syntax as an extension to the Rebeca syntax
- pRebeca model semantics based on Markov decision processes (MDPs)
- A method to model check pRebeca models by means of the PRISM model checker [9]

To verify pRebeca models, we use a two step method. In the first step pRebeca model is converted to an MDP by means of the state-space generation engine of Afra [10], a Rebeca model checker. In the second phase, the resulting MDP is expressed in the PRISM language and Probabilistic Computation Tree Logic (PCTL) properties can be checked against it. Hence, modelers using pRebeca can take advantage of using a high-level modeling language along with using PRISM features as a powerful model checking tool to verify their models.

The rest of this paper is organized as follows. Section 2 contains the related work and some explanation on differentiating points of this work from existing ones. Some preliminary definitions and concepts are explained in section 3. Section 4 consists of explanations of the syntax and semantics of pRebeca. Section 5 presents the method proposed to model checking pRebeca models and also includes a case study, and the work is concluded in section 6.

2 Related Work

There have been a number of process algebras [11–17], modeling languages, and model checking tools [9, 18–20] introduced so far to model and model check probabilistic systems. PRISM [9] is a well-known powerful probabilistic model checker with a guarded-command input language. A PRISM model is composed of reactive modules which can interact with each other. Although PRISM is a powerful model checker, the PRISM language does not support high-level constructs such as conditional or loop statements and just has a few primary data types, thus it is not suitable for high-level modeling of systems. ProbVerus [21] is another probabilistic model checker which is an extension to Verus verifier and implements symbolic techniques for PCTL model checking. ProbVerus does not support nondeterminism in its models. The ProbVerus language provides constructs such as assignment, conditional and loop statements and a probabilistic choice as a probabilistic feature and its semantics is based on Markov chains. PMAUDE is a specification language based on probabilistic rewriting theories with tool support for discrete event simulation and statistical verification. There is an implementation of actor modules in PMAUDE, in which the probabilistic distribution of time for message passing and computation is considered. The nondeterminism have been removed from the models for the sake of statistical model checking. Probmela [22] is a variant of Promela modeling language with tool support for quantitative Linear Temporal Logic (LTL) model checking. There has been also a direct mapping from Probmela to PRISM language. The Probmela Language constructs are close to our work in which there are high-level constructs such as conditionals, loops, probabilistic choices and primitives to model unreliable message passing.

The main difference of our work with others is that we extended an object based modeling language which is based on actor model and containing concepts such as classes and methods in Java-like style. pRebeca provides high-level constructs such as loops, conditional statements, assignment and also probabilistic primitives to model probabilistic choice between alternative behavior and unreliable message passing. All these facilities make pRebeca a suitable language for modeling asynchronous distributed systems. The method that we use for model checking keeps generated PRISM models in a rational size. This method prevents generation of models with extra overhead which may be resulted by direct mapping between two languages.

3 Preliminaries

3.1 Rebeca

Rebeca is a high-level modeling language based on actor model. The behavior of a Rebeca model is resulted from fairly interleaving of some self contained, reactive objects running on separate threads. Rebecs communicate only via asynchronous message passing and they have no shared state. Each rebec has an unbounded FIFO queue called its message queue to automatically receive messages. In Rebeca, rebecs are instances of reactive classes.

The structure of a Rebeca model consists of reactive class definitions and a "main" block. Each reactive class takes an integer value as an upper bound to its message queue which is used for model checking. There are three main parts in a reactive class:

- Definition of known rebecs denoting rebecs that can be receivers of the sending messages. Known rebecs define the infrastructure network topology.
- Definition of state variables that can be manipulated along processing of messages. State variables can be typed as integer, character, boolean or array.
- Definition of message servers which play the role of methods of the reactive class. The body of a message server contains a set of local variables and a sequence of statements. Messages in a Rebeca model are calls to message servers in the form of r.m(params) where r denotes one of the known rebecs of the sender, or **self** in case of sending a message to self. The message name m determines the message server of r that should be executed with the parameter list params. The statements in the body of a message server can be loops, conditional statements, assignment and (non-blocking) message send statements. Each rebec should have an **initial** message server which is invoked at the beginning of the model running. The execution of message servers is atomic. Thus, a computation step in a Rebeca model is the execution of a message server.

The main block in a Rebeca model consist of declaration of rebecs as the instances of reactive classes and binding of the declared rebecs to their known rebecs. Also, some parameter values can be sent to the initial message server of each rebec. At the beginning of the running of a model, instances of reactive classes are made and an invocation to the initial message server of each rebec is put into its message queue. Then, one of the rebecs is selected nondeterministically and the message at the head of its queue is popped out and executed and after that another rebec takes the turn and so on.

3.2 Markov Decision Process

In oreder to explain the semantics of pRebeca based on a Markov decision process we need to explain some definitions and concepts related to this structure. All the notations are adopted from [23].

Definition 1. *A distribution function over a set of states S can be denoted by* ρ*:* $S \rightarrow$ *[0,1] where* $\sum_{s \in S} \rho(s)=1$*. The set of all possible distribution functions over S is denoted by Distr(S).*

Definition 2. *A Markov decision process can be defined by a tuple* $M=(S,S_{init}$*, Act,* τ*,AP,L) where:*

- *S is a set of states.*
- S_{init} *is the set of initial states.*
- *Act is the set of actions.*
- $\tau \subseteq S \times Act \times Distr(S)$ *is the probabilistic transition relation.*
- *AP is the set of atomic propositions.*
- $L \subseteq S \times 2^{AP}$ *is the labeling function.*

In each state s, transitions take place by first nondeterministically choosing an action $a \in Act(s)$ where $Act(s)$ denotes the set of possible actions in state s. Then, there is a corresponding transition distribution function which determines the probability of the transition to other states in S. The transition relation $s \rightarrow^a \rho$ denotes that ρ is the distribution function in state s by selecting action a (simply, $\rho_{s,a}$ denotes the transition distribution function in state s by selecting action a). All $s' \in S$ for which $\rho(s')>0$ are called a-successors of s. A transition such as $s \rightarrow^a \rho_t^1$ denotes that the only a-successor of s is t and the probability of transition to t is one and this relation can be viewed as $s \rightarrow^a t$ in a Labeled Transition System.

4 pRebeca

In this section, we explain the syntax and semantics of pRebeca and provide an example in order to better understand the language constructs.

4.1 Syntax

The syntax of pRebeca is an extension to the syntax of the Rebeca language. In order to provide a concise syntax, we investigated possible probabilistic aspects that could exist in an actor based system. In the actor model, actors are executed independently and communicate just via asynchronous message passing. Thus, the probabilistic features in the actor model can be considered as follows:

- An actor can exhibit different alternative behaviors with some probability. Such as a node in the network which can behave differently in each of the safe or failure states.
- Messages can be lost being sent via unreliable communication media. Unreliability is a usual feature of communication media in the real world networks that should be considered in modeling.

To address the mentioned probabilistic features, we extended the Rebeca syntax by adding the following features.

– *Probabilistic alternative behavior:*
pAlt{
prob(p_1):{statement;*}
⋮
prob(p_n):{statement;*}
}
where $p_i \in [0,1]$, i ∈ {1,..., n}.

A pAlt statement denotes probabilistic choice between alternative behavior. In a pAlt structure, each block of statements may be executed by the mentioned probabilities. If the sum of the probabilities is less than one, the remaining probability indicates doing nothing with probability $p_{default}$=1-$\sum_{i=1}^{n} p_i$. As an example, with execution of the following structure:
pAlt {prob(0.3): {x=x+1} prob(0.6): {x=x-1;}}, the value of x may be incremented by one with probability 0.3 or it would be decremented by one with probability 0.6 and remain unchanged with probability 0.1.

– *Probabilistic assignment:*
x=?(p_1:v_1,. . . , p_n:v_n), where p_1+. . .+p_n=1 and v_1,..., v_n are possible values for the variable x.

A probabilistic assignment denotes assigning values v_1,. . ., v_n to x by respective probabilities p_1,. . ., p_n. As an example, the statement x=?(0.2:5, 0.8:2); assigns values 5 and 2 to x with probabilities 0.2 and 0.8 respectively. This primitive can be implemented by means of a **pAlt** construct but is included for the sake of simplicity, as a syntactic sugar.

– *Probabilistic message sending:*
r.m() **probloss**(p)

This statement denotes sending messages with loss probability of p. This feature is added in order to model unreliable communications. As an example, the execution of r.m()probloss(0.2) may result in adding a message to the message queue of rebec r with probability 0.8 and no changes would happen with probability 0.2.

The grammar of pRebeca is presented in Figure 1. Complete explanations on the syntax of the Rebeca language can be found in [1].

As client-server architecture is one of the common architectures used for distributed systems, we choose an unreliable client server system as a running example. In this model, there are two reactive classes Client and Server. Three instances of Client and one instance of Server are declared. All clients can send requests to the server and do not communicate directly. Instances of Server have two operating modes: *safe* mode in which the server exhibits its expected functionality and *failure* mode in which the server is unable to respond to the requests correctly. A client exhibits probabilistic behavior in the way that it

Model::= reactiveClass* Main
reactiveClass::= **reactiveclass** C(I){knownRebecs stateVars msgSrv*}
Main::= {rebecDcl*}
knownRebecs::= **knownrebecs**{krDcl*}
msgSrv::= **msgsrv**(<T v>*){stmt*}
stateVars::= **statevars**{varDcl*}
varDcl::= T <v>+;
krDCL::= C c;
stmt::= v=e;|r=new C(<e>*);| call;| conditional| pAlt| pAssignment
call::= r.M(<e>*)|r.M(<e>*)**lossprob(p)**
conditinal::= if(e){stmt+}| if(e){stmt+} else{stmt+}
rebecDcl::= C r(<r>*):(<c>*);
pAlt::= **pAlt**{prob(p_1){stmt+ },..., prob(p_n){stmt+}}
pAssignment::= v=?(<p_i:c>+);

Fig. 1. BNF grammar for pRebeca language. Superscript + denotes repetition of one or more times and superscript * denotes zero or more times repetition. Using angle brackets with repetition denotes comma separated lists. The Symbols C, T, I, v, c, r, m, and e denote reactive class, type, constant, variable, class name, rebec, message server, and expression respectively. Expressions in pRebeca are the same as expressions in Java.

chooses one of the existing client identifiers probabilistically and then, sends a message containing the selected id to the server. The number of repetitions of this message sending is determined via another probabilistic choice. Clients ignore the received messages which have different receiver ids. The server checks the id in the message and if in safe mode, sends the message to the corresponding id. If it is in the failure mode, it either ignores the message or sends it to a wrong client. The communication media in this system is lossy. This example does not describe any specific and realistic system and is just used for the sake of better representing the syntax and semantics of the language. The pRebeca model of unreliable client-server system is presented in Figure 2.

4.2 Semantics

Before explaining the semantics of a pRebeca model we need to define some notations and basic given sets.
In a pRebeca model P, we assume to have the following sets.

- Id_P is the set of rebec identifiers.
- Val_P is the set of all possible values for all state variables.
- MS_P is the set of all message servers in the model.

```
1 reactiveclass Server() {
2   knownrebecs { Client cl1,cl2,cl3; }
3   statevars { boolean mode; }
4   msgsrv initial() {}
5  msgsrv takeRequest(int Id) {
6   mode=false;
7   pAlt{
8    prob(0.7){ //server in safe mode
9        mode=true;
10       if(Id==1)
11            cl1.receive(Id)probloss(0.3);
12       if(Id==2)
13            cl2.receive(Id)probloss(0.2);
14       if(Id==3)
15            cl3.receive(Id) probloss(0.4);}
16    prob(0.2){ //server in failure mode
17      if(Id==1){
18            cl2.receive(Id)probloss(0.2);
19            cl3.receive(Id)probloss(0.4);}
20      if(Id==2){
21            cl1.receive(Id)probloss(0.3);
22            cl3.receive(Id)probloss(0.4);}
23      if(Id==3){
24            cl1.receive(Id) probloss(0.3);
25            cl2.receive(Id) probloss(0.2);}
26    }
27  }
28 }
29 main{
30 Server Server1(client1,client2,client3):();
31 Client client1(server1):(1,2,3);
32 Client client2(server1):(2,1,3);
33 Client client3(server1):(3,2,1);
34 }
```

```
1'  reactiveclass Client() {
2'    knownrebecs {Server mServer;}
3'    statevars {
4'          int rand,Id1,Id2,myId,rep;
5'          boolean ignore;}
6'   msgsrv initial(int m,int n,int k){
7'     myId = m;
8'     Id1 = n;
9'     Id2 = k;
10'     self.send();
11'   }

12'   msgsrv send() {
13'       rand=?(0.3:Id1,0.7:Id2)
14'       rep=?(0.5:1,0.5:2)
15'       while(rep>0)
16'       {
17'          mServer.takeRequest(rand)
          probloss(0.2);
18'         rep−−;
19'       }
20'       self.send();
21'   }
22'   msgsrv receive(int recId){
23'       If(recId!=myId)
24'       ignore=true;
25'   }
26' }
```

Fig. 2. The pRebeca model of unreliable client-server system

We formalize the structure and state of a rebec in a pRebeca model P, as follows:

Definition 3. *The structure of a rebec, r_i (where $r_i \in Id_P$), can be described by a tuple (SV_i, KR_i, MS_i) where:*

- *SV_i is the set of the state variables of r_i.*
- *KR_i is the set of the known rebecs of r_i.*
- *MS_i is the set of the message servers in r_i.*

Definition 4. *The structure of a message received by rebec r_i can be formalized by a tuple Msg=(senderId, M) where:*

- *$senderId \in Id_P$.*
- *$M \in MS_i$.*

We denote the message queue of rebec r_i by q_i and define the following functions for a message queue:

Definition 5. *head(q_i) denotes the message at the head of message queue q_i. append(q_i, msg) denotes adding msg to the end of message queue q_i.*

Definition 6. *The state of a rebec r_i, can be defined by means of an evaluation function s_i: $SV_i \rightarrow Val_P$. The function s_i determines the value of each state variable in r_i at the current state.*

We separate the contents of the message queue of a rebec from its state for the sake of better matching and explaining the semantics based on an MDP structure. Hence, we define an evaluation function determining the content of the message queues in the model.

Definition 7. *The message queue evaluation function is defined as the function $\zeta:(\bigcup q_i) \rightarrow (Id_P \times MS_P)^*$ where q_i denotes the message queue of r_i and $(Id_P \times MS_P)^*$ denotes all possible sequences of messages in a message queue.*

The operational semantics of a pRebeca model is based on an MDP in which, each state is a combination of the states of the rebecs included in the model and the message queue evaluation function. The transitions in this MDP can take place by first, nondeterministic selection of one of the rebecs with nonempty message queue and popping out the first message which determines the action of the transition, and then, a distribution function corresponding to the action determines the probability of transitions to the succeeding states.

Definition 8. *The operational semantics of a pRebeca model, P with n rebecs, can be defined as an MDP M_P=(S_P, S_{0P}, Act_P, τ_P, AP_P, L_P) where:*

- *S_P is the set of states where each state is a tuple of the form:*

$$(s_1, ..., \ s_n, \ \zeta) \tag{1}$$

  where s_i denotes the state of rebec i and ζ is the evaluation function of message queues.
- *S_{0P} is the initial state where all rebecs are in their initial states $s_{init\ i}$. In the initial state of each rebec, all the state variables have their initial values and the only message in each rebec's message queue is an initial message.*
- *$Act_P \subseteq Id_P \times MS_P$ is the set of actions in M_P which is the set of all possible messages that can be sent in P.*
- *AP_P is the set of atomic propositions which is a subset of $Id_P \times (\bigcup SV_i)$. We consider the Boolean-valued variables of the rebecs as labels. The atomic proposition (r_i, v) corresponds to variable v of rebec r_i.*
- *L_P is the labeling function $S_M \rightarrow 2^{AP_P}$, which assigns the variables with true value to the states.*
- *$\tau_P \in S_P \times ACT_P \times Dist(S_P)$ is the set of transitions in M_P. A transition in M is defined by:*

$$\frac{head(q_i) = a_i}{(s_1, ..., \ s_n, \ \zeta) \rightarrow^{a_i} \upsilon} \tag{2}$$

  where s_i is the state of rebec r_i and a_i is an invocation of one of the message servers of rebec r_i. This formula denotes the nondeterministic selection of action a_i at state (s_1,..., s_n, ζ). the function υ is the distribution function determining the probability of transition to the a_i-successors of state (s_1,..., s_n,

ζ). The probability of transitions is determined from the effect of statements in the body of the corresponding message server (as will be described shortly).

Processing action a_i can only make changes to the state of rebec r_i and also can change the contents of the message queues in the model. As in a pRebeca model a message is an invocation of a message server and the execution of message servers is atomic we need to specify the effect of each statement separately and afterwards the effect of executing a sequence of statements.
Assignment, conditional statements and loop statements have the same meaning as in Rebeca and a complete explanation of the semantics of these statements can be found in [1]. Thus, we continue with explaining the semantics of probabilistic primitives of the pRebeca language only. As said before, processing a message by rebec r_i only affects s_i and ζ. Hence, in the following we restrict our notation to rebec r_i only. To be more precise, all of the notations defined bellow must be subscripted by i, but we drop the subscript to make it more readable.

Definition 9. *The set of local states combined with the message queues is defined as IState= $(SV_i \rightarrow Val_P) \times (\bigcup q_i \rightarrow (Id_P \times MS_P)^*)$.*
For $\sigma=(s \times \zeta) \in IState$, s denotes the state of r_i and ζ is the contents of the queues.

Definition 10. *A distribution function specifying the probability of transition to the successor IStates can be defined as: φ: IState$\rightarrow$[0,1].*

In order to recursively compute the probabilities of transitions from an IState after the execution of a sequence of statements, we need to define the sum of two distribution functions in the case that equal IStates are generated from different paths of computations which may be a result of different combination of probabilistic choices during the execution of probabilistic statements.

Definition 11. *If $\varphi=\{\sigma_1 \mapsto p_1,..., \sigma_k \mapsto p_k\}$ and $\varphi'=\{\sigma'_1 \mapsto p'_1,..., \sigma'_k \mapsto p'_k\}$ be two distribution functions over IStates, then sum of the functions is defined as:*
$\varphi+\varphi'=\{\sigma \mapsto p \mid \sigma \in \{\sigma_1,..., \sigma_k\} \bigcup \{\sigma'_1,..., \sigma'_k\}\}$
where $p=\frac{\varphi(\sigma)+\varphi'(\sigma)}{TotalProb}$, $TotalProb=\sum p$, $p \in \{p_1,..., p_k, p'_1,..., p'_k\}$. We assume $\varphi(\sigma_j)=0$ for σ_j which is not in the domain of φ and $\varphi'(\sigma_j)=0$ for σ_j which is not in the domain of φ'. We also use the notation: $\sum_{j=1}^{n} \varphi_j=\varphi_1+...+\varphi_n$.

We also define the scalar product of a probability p $\in$ [0,1] in a distribution function as follows:

Definition 12. *If $\varphi=\{\sigma_1 \mapsto p_1,..., \sigma_k \mapsto p_k\}$ and $p \in [0,1]$ then $p.\varphi=\{\sigma_1 \mapsto p.p_1,..., \sigma_k \mapsto p.p_k\}$.*

Definition 13. *The effect of a statement stmt on an IState σ is defined by means of the function Effect: IState $\times$ stmt $\rightarrow$ (IState $\rightarrow$ [0,1]). In fact, Effect(σ, stmt)=$(\sigma_1 \rightarrow p_1,..., \sigma_k \rightarrow p_k)$ where σ_i is one of the possible successors of σ with transition probability p_i.*

Definition 14. *The effect of a sequence of statements is defined by Effect(σ, stmt;stmtlist)=$\sum$ p_j.Effect(σ_j, stmtlist), $1 \leq j \leq k$, where Effect(σ, stmt)=($\sigma_1 \mapsto p_1$,..., $\sigma_k \mapsto p_k$) and stmtlist::= stmt | stmt;stmtlist denotes a sequence of statements.*

Now we define the effect of the probabilistic statements in pRebeca model:

- *Effect*(σ, pAlt{ prob(p_1): {stmtlist$_1$},..., prob(p_n): {stmtlist$_n$}})= $\sum$ p_j.Effect(σ, stmtlist$_j$).

- *Effect*(σ,x=?(p_1: v_1,..., p_n: v_n))=(σ[x:=v_1]$\mapsto$ p_1,..., σ[x:=v_n]$\mapsto$ p_n) where σ[x:=v_i] denotes a successor state of σ in which the value of the state variable x is changed to v_i.

- *Effect*(σ, r_i.m()probloss(p))=(σ[q_i=append(q_i, m( ))] $\mapsto$ (1-p), $\sigma \mapsto$ p) where σ[q_i=append(q_i, m())] denotes the state which differs from σ only in the contents of message queue q_i where a new message m is added to the end of the message queue.

Now we can explain the transition relation in the mentioned MDP. In transition (s_1,..., s_n,ζ)$\rightarrow^{a_i}$ υ where rebec r_i processes message a_i, we have:

-υ($s_1^{'}$,..., $s_n^{'}$, $\zeta^{'}$)=0 for the states where $s_j^{'} \neq s_j$ for i $\neq$ j.

-υ($s_1^{'}$,..., $s_n^{'}$, $\zeta^{'}$)=*Effect*(σ, Body(a_i)) where $s_j^{'} = s_j$ for i $\neq$ j.

where σ=(s_i, ζ) and Body(a_i) denotes the sequence of statements of the message server invoked by a_i.

5 Model Checking pRebeca

In this section, we explain our method for model checking pRebeca models. We chose PRISM as the model checker for pRebeca models since it is a powerful tool supporting analysis of models based on MDPs and other Markovian structures. We consider two different methods to analyze pRebeca models by means of PRISM. The first is the direct mapping of pRebeca models to PRISM models which is very complicated since the PRISM modeling language does not support most of the required high level constructs both in data and control aspects. There are many entities in a pRebeca model that can not be easily implemented in the PRISM language such as message queues and loops.

The PRISM language does not support array data type. Thus we need to model a message queue by defining a number of variables which model the locations of the queue and a variable to keep the number of messages in the queue. Popping out a message from a message queue in pRebeca which is automatically

done can be modeled in PRISM by shifting the value of all variables towards the head and update the value of variable that keeps the number of filled elements. This part of code should be rewritten for different possible number of messages in a message queue. Besides, there is another problem with keeping parameters that can be sent through a message because there is no support for data structures in PRISM. There would be similar problems in implementing other high level constructs of pRebeca by PRISM language. The direct mapping as explained above may require to define a large number of variables which may cause extraordinary growth in model size and makes the model checking impossible.

As the direct mapping of pRebeca models to PRISM models would be overbearing and may cause extraordinary growth in model size and makes the model checking impossible we provided a two step method which makes significantly less overhead. This method has two main steps that can be explained as follows:

1. The pRebeca model is converted to an MDP using state space engine of Afra which is a Rebeca model checker.
2. The generated MDP is converted to a PRISM model.

The result of the first step is an MDP which contains the states, transitions, and the probability of each transition. Before explaining the description of an MDP by means of the PRISM language, we give a brief explanation about PRISM language. A command in the PRISM language is in the form:
$[\,]\text{guard} \rightarrow \text{prob}_1 : \text{update}_1 + ... + \text{prob}_n : \text{update}_n;$

If the guard holds, each one of the updates may take place by the corresponding probability. If the guards of two or more commands hold simultaneously, one of them is executed nondeterministically.

In order to describe an MDP structure in PRISM, we assign each state of the MDP a number and the probability of transitions would be the corresponding probabilities with update parts. An update is the change of current state number which shows transition to another state. Figure 3 represents the transformation of a part of the state space of our running example, generated by Afra, in the form of an MDP and the corresponding PRISM specification. Using this method based on our experience has less overhead and keeps the size of the resulting models in an acceptable order to be checked by PRISM.

5.1 Case Study

We choose asynchronous leader election algorithm based on [24] as a case study. Leader election is one of the well-known algorithms used in distributed systems. In this variant of the algorithm, the nodes do not have any identifiers and are identical, spread over a network. In our case study, the network has ring topology, however the network topology can be easily changed by making a few changes to known rebecs part. Each node has a left and a right neighbour and only sends

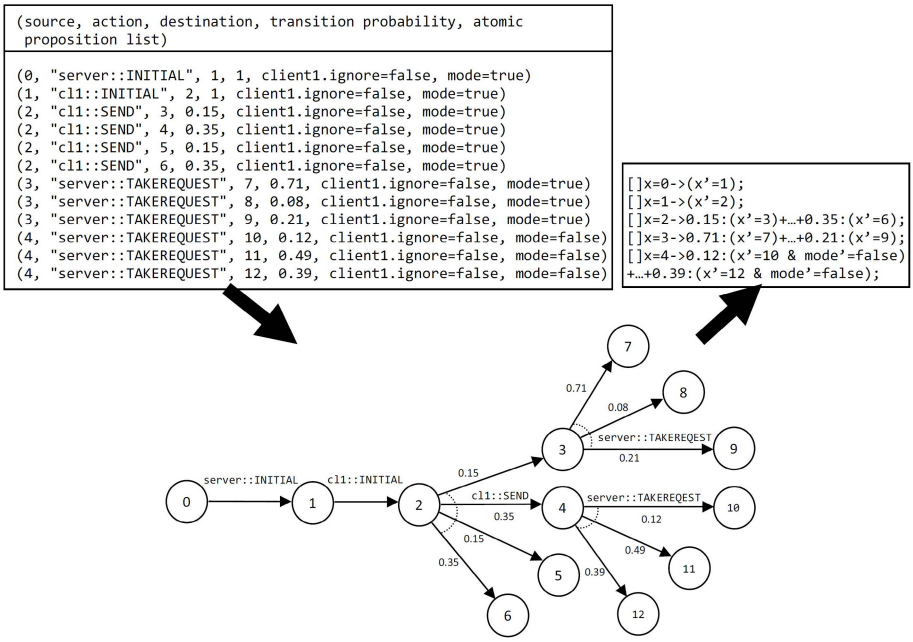


Fig. 3. Transforming pRebeca transitions (top-left) to PRISM specification (top-right) based on the equivalent MDP (bottom)

messages to its right neighbour. There are two modes for each node, active and inactive. At the beginning, all nodes are active. To elect a leader at the starting point, each node flips a coin and sends the result to its right neighbour. If a node has a head (denoted by false) in flipping coin and its left neighbour gets tail (denoted by true) it becomes inactive, and it repeats the process otherwise. An inactive node can only pass messages it receives to its right neighbour. Nodes initially have the number of active nodes and in the case that a node becomes inactive, it decreases the number of active nodes by one. The number of active nodes will be passed through messages making it possible for nodes to update their local information. The execution will continue until the number of active nodes equals 1. The pRebeca code of leader election protocol is presented in Figure 4.

We have modeled this system for 3, 4 and 5 nodes and the resulting PRISM models have the proper size to be checked by the PRISM model checker. We have model checked some PCTL properties such as the maximum probability of electing a leader in 2N rounds where N is the number of nodes and the maximum probability of finally electing a leader. We have also analyzed a number of well-known protocols and algorithms such as randomized dinning philosophers (with 6 philosophers) and IPV4 zeroconf protocol and the time/space consumption of model checking was reasonable.

```
1 reactiveclass Node() {
2   knownrebecs {
3     Node rNeighbour;
5   }
6   statevars {
7     boolean leader;
8     int nActive;
9     boolean active;
10     boolean x;
11   }
12   msgsrv initial(int num) {
13     nActive=num;
14     active=true;
15     leader=false;
16     self.flip();
17   }
18   msgsrv flip() {  //flipping the coin and sending result to right neighbour
19     x=?(0.5:false,0.5:true);
20     rNeighbour.elect(x,nActive);
21   }
22   msgsrv elect(boolean i,int n) {
23     if(active==true){
24               if(nActive==1){
25                       leader=true;
26                       rNeighbour.leaderElected()
27               }
28               else{
29                       if(nActive>n)  //updating the number of active nodes if needed
30                               nActive=n;
31                       if(x==false ∧ i==true){
32                               active=false;
33                               nActive−−;
34                       }
35                       else{
36                               self.flip();
37                       }
38               }
39     }
40     else{  //just passing received messages in case of being inactive
41                 rNeighbour.elect(i,n);
42     }
43   }
44   msgsrv leaderElected(){
45         if(leader==false){  //informing right neighbours that a leader has been elected
46               active=false
47               rNeighbour.leaderElected();
48         }
49   }
50 }
51 main{  //declaration of rebecs and passing the number of rebecs to initial message servers
52       Node n0(n1):(3);  //the number of nodes is passed to initial message servers
53       Node n1(n2):(3);
54       Node n2(n0):(3);
55 }
```

Fig. 4. The pRebeca model of randomized leader election

6 Conclusion and Future Work

In this paper, we presented pRebeca, an extension to an object-based high-level modeling language based on actor model, which is suitable for modeling asynchronous distributed systems. We proposed the syntax of pRebeca which is an extension of the Rebeca syntax and also provided the semantics of this language based on MDPs. A two step method was presented to model check pRebeca models by means of the well-known probabilistic model checker PRISM. Using the proposed approach, the modeler of a probabilistic distributed system can effectively model the system in a high-level, readable, and maintainable language. These benefits all come from the object encapsulation and elegant concurrency paradigm in actor model, as well as familiar, high-level Java-like syntax of Rebeca. Our effective way to generate PRISM models from pRebeca enables to have the mentioned benefits, while using a powerful model checking engine like PRISM. Although our method reduces the complexity of converting pRebeca model to a proper input model described by the PRISM language, we are planning to develop methods and tools to better and more efficiently analyze pRebeca models.

References

1. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inf. 63(4), 385–410 (2004)
2. Hewitt, C.: Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot. MIT Artificial Intelligence Technical Report 258, Department of Computer Science. MIT (April 1972)
3. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
4. Agha, G., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7, 1–72 (1998)
5. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the JVM platform: a comparative analysis. In: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, pp. 11–20. ACM, New York (2009)
6. Hewitt, C.: Orgs for scalable, robust, privacy-friendly client cloud computing. IEEE Internet Computing 12(5), 96–99 (2008)
7. Hewitt, C.: Actorscript(tm): Industrial strength integration of local and nonlocal concurrency for client-cloud computing. CoRR abs/0907.3330 (2009)
8. Agha, G., Meseguer, J., Sen, K.: Pmaude: Rewrite-based specification language for probabilistic object systems. Electron. Notes Theor. Comput. Sci. 153(2), 213–239 (2006)
9. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
10. Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: Sysfier: Actor-based formal verification of systemc. ACM Trans. Embed. Comput. Syst. 10(2), 19:1–19:35 (2011)

11. Baier, C., Kwiatkowska, M.Z.: Domain equations for probabilistic processes. Electr. Notes Theor. Comput. Sci. 7, 34–54 (1997)
12. den Hartog, J., de Vink, E.P.: Mixing up nondeterminism and probability: a preliminary report. Electr. Notes Theor. Comput. Sci. 22, 88–110 (1999)
13. Hansson, H.A.: Time and Probability in Formal Design of Distributed Systems. Elsevier Science Inc., New York (1994)
14. Larsen, K.G., Skou, A.: Compositional Verification of Probabilistic Processes. In: Cleaveland, W.R. (ed.) CONCUR 1992. LNCS, vol. 630, pp. 456–471. Springer, Heidelberg (1992)
15. Lowe, G.: Probabilistic and prioritized models of timed csp. Theor. Comput. Sci. 138(2), 315–352 (1995)
16. Penczek, W., Szałas, A. (eds.): MFCS 1996. LNCS, vol. 1113. Springer, Heidelberg (1996)
17. Tofts, C.: A Synchronous Calculus of Relative Frequency. In: Baeten, J.C.M., Klop, J.W. (eds.) CONCUR 1990. LNCS, vol. 458, pp. 467–480. Springer, Heidelberg (1990)
18. Ciesinski, F., Baier, C.: Liquor: A tool for qualitative and quantitative linear time analysis of reactive systems. In: Proc. 3rd International Conference on Quantitative Evaluation of Systems (QEST 2006), pp. 131–132. IEEE CS Press (2006)
19. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The Ins and Outs of The Probabilistic Model Checker MRMC. In: Quantitative Evaluation of Systems (QEST), pp. 167–176. IEEE Computer Society (2009), www.mrmc-tool.org
20. Bohnenkamp, H.C., D'Argenio, P.R., Hermanns, H., Katoen, J.P.: Modest: A compositional modeling formalism for hard and softly timed systems. IEEE Trans. Software Eng. 32(10), 812–830 (2006)
21. Katoen, J.-P. (ed.): ARTS 1999. LNCS, vol. 1601. Springer, Heidelberg (1999)
22. Proceedings of the 2nd ACM & IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE 2004), San Diego, California, USA, June 23-25. IEEE (2004)
23. Baier, C., Katoen, J.P.: Principles of Model Checking. Representation and Mind Series. The MIT Press (2008)
24. Itai, A., Rodeh, M.: Symmetry breaking in distributed networks. Information and Computation 88(1) (1990)

Modular Verification of OO Programs with Interfaces*

Qiu Zongyan[1,2], Hong Ali[1], and Liu Yijing[1]

[1] LMAM and Department of Informatics, School of Math., Peking University, China
[2] State Key Laboratory of Computer Science, ISCAS, China
{qzy,hongali,liuyijing}@math.pku.edu.cn

Abstract. *Interface types* in OO languages support polymorphism, abstraction and information hiding by separating interfaces from their implementations. The separation enhances modularity of programs, however, it causes also challenges to the formal verification. Here we present a study on interface types, and develop a specification and verification theory based on our former VeriJ framework. We support multi-specifications for classes inherited from interfaces and the superclass, and keep the verification modularly without re-touching the verified code. The concepts developed in VeriJ, namely the *abstract specification* and *specification predicate*, play important roles in this extension, and thus are proved widely useful and very natural in the formal proofs of OO programs.

1 Introduction

The reliability and correctness for software systems attract more and more attentions, because faults in an important system may cause serious damages or even loss of life. Object-Oriented (OO) techniques are widely used in software practice, and thus the useful techniques supporting high-quality OO development are really demanded, e.g., the formal verification techniques. Core OO features, saying modularity, encapsulation, inheritance, polymorphism, etc., enhance scalability of programs greatly in practice, but they bring also challenges to formal verification. Encapsulation implies information hiding and invisible (and replaceable) implementation details; polymorphism enables dynamically determined behaviors. Both cause difficulties to the verification.

Separating the interface from implementation is one of the most important techniques in OO development. With this separation, clients can use objects of different types via a common interface, and call methods determined by concrete types of the objects. It enables low coupling of clients from implementations, and makes the system easy to modify and extend. The technique is used widely, e.g., it appears in many design patterns [6]. To support this important technique, many OO languages, notably Java and C#, offer special features. In languages without direct support, various features to mimic interfaces are available. To verify OO programs with interfaces raises new challenges. We need to define formally the roles played by interfaces, and make clear not only relations among them, but also their relations with classes. We should develop ways for verifying client code which uses objects via variables of interface types. Note that interfaces provide only method declarations without implementations.

* Supported by NNSF of China, Grant No. 90718002 and 61100061.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 151–166, 2012.

How to specify these methods independently for verification makes a new challenge. In addition, we want modular verification: neither re-verifying class implementations, nor touching implementation details of the classes in verifying client code.

Behavioral subtyping [13] is one of the most important concepts in OO world, a cornerstone in OO practice, and a crucial element of many formal works for OO, e.g. [2,9]. An OO program obeying behavioral subtyping gives good support to reason it modularly, because it demands that an object of a subtype behaves the same when it is used in a context where a supertype object is required. However, when multiple interfaces present, how can we define behavioral subtyping without implementations?

Many concepts have been proposed for the verification in OO area, e.g., *model field/abstract field* [3,11,14], *data group* [12], and *pure methods* [17], etc. Verifying OO programs with interfaces is also studied in some work, e.g. [16,14,7,2,5]. [16] integrated interface types and proposed some techniques which inspired many later research. The work on JML and Spec# [7,2] introduced specifications for interface types with data abstraction to some extent, ensured behavioral subtyping and developed some verification tools. However, these early work have some weak points. For example, as pointed by [15], none of them addresses the inheritance in a satisfactory way, because they either restrict behaviors of subclasses, or require re-verification of inherited methods. In addition, mutable object structures are largely neglected. To remedy the situation, [15,4] provided similar ideas by suggesting dual specifications for a method, where the *static* one describes its detailed behavior for verifying implementation, and the *dynamic* one supports verifying dynamically bound invocations. However, this dual is not necessary, and our VeriJ framework [18] can handle the problem where only one specification for each method is provided.

To develop a verification framework for OO languages with interface types is the goal of this study. It seems not easy to extend the *dual specification* approaches to cover interface types, because interfaces do not have behaviors, neither static nor dynamic. Our VeriJ framework is based on *abstract specifications* and *specification predicates*, while the former supports specifying method behavior on a suitable abstract level, and the latter serves to connect the specification with concrete implementations. Based on these concepts, we develop a framework which seems very satisfactory.

Our main contributions are: (1) we give a deep analysis for interface types when the verification is the goal; (2) we define a framework to support abstract specification for interface-based information hiding and encapsulation. We show that the *specification predicate*s are natural in connecting abstract specification with implementation details, and supporting modular verification of different implementing classes under an interface; (3) we propose a general model for dealing with multi-inheritance of specifications, and inference rules for proving the implementation and client code modularly; (4) we develop some examples to show the power and usefulness of our framework. To our limited knowledge, this is the first framework which can avoid reverification of method bodies in a language with rich OO features and interface types.

In Section 2, we discuss key situations that a useful theoretical work for interface types must address. We introduce briefly our assertion language and VeriJ in Section 3, and define its verification framework in Section 4. We illustrate our ideas for modular specification and verification by an example in Section 5. Then we conclude.

$$\textbf{inter}\ I_1\ \{\ T_1\ m(..);\ \ T_2\ f(..);\ \}\qquad \textbf{inter}\ I_2\ \{\ T_1\ m(..);\ \ T_3\ g(..);\ \}$$
$$\textbf{class}\ \ B : \textbf{Object}\ \{\ T_1\ m(..)\{...\}\ \ T_3\ g(..)\{...\}\ \ T_4\ h(..)\{...\}\ \ T_5\ k(..)\{...\}\ \}$$
$$\textbf{class}\ \ D : B \rhd I_1, I_2\ \{\ T_1\ m(..)\{...\}\ \ T_2\ f(..)\{...\}\ \ T_5\ k(..)\{...\}\ \ T_6\ n(..)\{...\}\ \}$$
$$\textbf{class}\ \ E : \textbf{Object} \rhd I_1\ \{\ T_1\ m(..)\{...\}\ \ T_2\ f(..)\{...\}\ \ T_7\ p(..)\{...\}\ \}$$

Fig. 1. A Program with Interfaces

2 Interfaces and Verification: Basics

To give some ideas for the problem, code in **Fig. 1** is used in the discussion. The basic language we use is similar to Java or JML with some abbreviations for saving space. We will present some issues related to the specification and verification of OO programs with interfaces. We take "type" as a generic word for either a class or an interface.

Here we declare two interfaces I_1 and I_2, while each declares some method prototypes. Different interfaces may declare methods with the same name, e.g. m here. Here are also three classes. B takes **Object** as its superclass and defines some methods. It implements neither I_1 nor I_2. Class D is defined as B's subclass, which inherits g, h from B, overrides m, k of B, and defines f and n itself. In addition, D implements both I_1 and I_2, thus it must implement (and has implemented) all methods declared in I_1 and I_2. There are some interesting phenomena: D implements f of I_1 itself, but inherits g from B to implement g of I_2. The situation for method m is more complex. Here both interfaces declare a method prototype with name m, in addition, a method with the same name is defined in B too. In D, a new definition overrides m in B and implements the m in both I_1 and I_2. At last, another class E implements also I_1.

We think that an interface defines a type, while a class defines a type with an implementation. When a class implements some interfaces or inherits a superclass, it defines a subtype of them. To simplify the discussion, we assume that data fields in classes are all protected, and thus are not visible out of their classes. Under this assumption, a type is just a set of methods with signatures. We must have some type-related constraints. As in Java, when a class implements an interface I, it must provide all implementations for the methods declared in I, by either defining in itself or inheriting from its superclass. When a class implements several interfaces, a method with multiple declarations in these interfaces and/or the superclass must have the same signature, e.g. m in our example code. Constraints like these should be checked to ensure well-formedness. In the following, we suppose all programs under discussion to be well-formed.

For verification, we need specifications for methods. Assuming method m in D has specification $\pi_{_D}$, we need to prove that the implementation of m in D satisfies $\pi_{_D}$. In addition, we should support verifications of client code which calls methods. For example, here we may need to verify programs as:

$$I_1\ x = \textbf{new}\ D(..);\ I_2\ y = (D)x;\ \ \ldots\ x.m(..);\ y.m(..); \tag{1}$$

Here a D object is created and assigned to variable x of type I_1, and then to variable y of type I_2. Afterward, method m is called from x and y respectively, where the D object is used from variables of types I_1 and I_2. To verify the code, we hope to only consult

information for m in I_1 or I_2, but neither its implementation nor its specification in D. This restriction is clear, because there may be another creation, e.g.,

$$I_1\ z = \mathbf{new}\ E(..);\ x = (E)z;$$

and then the control may go to the same call statement from x in (1). This means that methods in interfaces must go with their specifications to support verification of client code as (1). The method declarations in interfaces have no body, thus their specifications must be abstract and say nothing about the implementation.

We extend method declarations and definitions in interfaces/classes as follows:

$$T_1\ m(..)\ \langle\varphi\rangle\langle\psi\rangle; \qquad T_1\ m(..)\ \langle\varphi\rangle\langle\psi\rangle\ \{\ \ldots\ \}$$

Here φ and ψ are the pre/post conditions of m, which are specified by **requires** and **ensures** clauses in JML/SPEC# respectively. For interfaces, we need to specify a method based on its calling object, parameters and return only, because no body here.

To support modular verification, a class C should be not only a behavioral subtype of its superclass, but also a behavioral subtype of its implementing interfaces, because C objects may be used via variables of any of C's supertypes. In our example, D must be a behavioral subtype of B, I_1 and I_2. Suppose the specifications of m in I_1, I_2, and B are π_1, π_2, and $\pi_{_B}$ respectively, then we must prove that m's implementation in D satisfies these three specifications. As a simplification, we may prove some correct relations between $\pi_{_D}$ with each of π_1, π_2, and $\pi_{_B}$.

If a method is explicitly specified, its specification is obvious. We support specification inheritance in our framework as in JML and Spec#. This means, when a method is not explicitly specified, its specification is inherited from its supertype(s). Because one class can have more than one supertypes (some interfaces and one superclass), a method may inherit several specifications. For example, if m in D is not explicitly specified, then it inherits specifications π_1, π_2, and $\pi_{_B}$. In this case, we take its specification as a set $\{\pi_1, \pi_2, \pi_{_B}\}$. Then we should define how a method body satisfies a specification as this, and how this specification is used in verifying client code.

Here we have provided an outline for the problems when we think about the verification of OO programs with interfaces. Based on our previous work VeriJ [18], with the concepts of abstract specification and specification predications, we develop a modular verification framework for these OO programs.

3 VeriJ: An OO Language with Specifications

Now we introduce our specification and programming language used in the work.

In [19] we developed OO Separation Logic (OOSL) for describing OO states and reasoning about programs. We use it as the assertion language in this work. Here we give a short introduction to OOSL. Readers can refer [19] to find more details.

OOSL is similar to the Separation Logic with some revisions:

$$\begin{array}{l}\rho ::= \mathbf{true} \mid \mathbf{false} \mid r_1 = r_2 \mid r : T \mid r <: T \mid v = r\\ \eta ::= \mathbf{emp} \mid r_1.a \mapsto r_2 \mid \mathsf{obj}(r, T)\\ \psi ::= \rho \mid \eta \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \psi \Rightarrow \psi \mid \psi * \psi \mid \psi -\!\!* \psi \mid \exists r \cdot \psi \mid \forall r \cdot \psi\end{array}$$

$$
\begin{array}{rcl}
v & ::= & \mathbf{this} \mid x \\
e & ::= & \mathbf{true} \mid \mathbf{false} \mid \mathbf{null} \mid v \mid d \mid e \odot e \\
b & ::= & \mathbf{true} \mid \mathbf{false} \mid e < e \mid e > e \\
 & \mid & e = e \mid \neg b \mid b \vee b \mid b \wedge b \\
c & ::= & \mathbf{skip} \mid x := e \mid v.a := e \mid x := v.a \mid x := (C)v \\
 & \mid & x := v.m(\overline{e}) \mid x := \mathbf{new}\, C(\overline{e}) \mid \mathbf{return}\ e \\
 & \mid & c; c \mid \mathbf{if}\ b\ c\ \mathbf{else}\ c \mid \mathbf{while}\ b\ c \\
T & ::= & \mathbf{Bool} \mid \mathbf{Object} \mid \mathbf{Int} \mid C \mid I \\
\pi & ::= & \langle\varphi\rangle\langle\psi\rangle
\end{array}
$$

$$
\begin{array}{rcl}
M & ::= & T'\ m(\overline{T_1\ z}) \\
P & ::= & \mathbf{def}\ [\mathbf{pub}]\ p(\mathbf{this}, \overline{x}) \\
L & ::= & \mathbf{inter}\ I\ [:\ J]\ \{\overline{P};\ \overline{M\ [\pi]};\} \\
K & ::= & \mathbf{class}\ C : B\ [\triangleright \overline{I}]\ \{ \\
 & & \quad \overline{[\mathbf{pub}]\ T\ a};\ \overline{P : \psi}; \\
 & & \quad \overline{C(\overline{T_1\ z})\ [\pi]\ \{\overline{T_2\ y};\ c\}} \\
 & & \quad \overline{M\ [\pi]\ \{\overline{T_2\ y};\ c\}}\ \} \\
G & ::= & (K \mid L) \mid (K \mid L)\ G
\end{array}
$$

Fig. 2. Syntax of VeriJ

where T is a type, v a variable or constant, r_1, r_2 references, ψ an assertion:

- ρ denotes assertions which are independent of heaps. For any references r_1 and r_2, $r_1 = r_2$ iff r_1 and r_2 are identical. $r : T$ means that r refers to an object with exact type T. $r <: T$ means that r refers to an object of T or one of its subtypes. And $v = r$ asserts that the value of variable or constant v is r.
- η denotes assertions involving heaps: $\mathbf{emp}$ asserts an empty heap; the singleton assertion $r_1.a \mapsto r_2$ means that the heap is exact a field a of an object (denoted by r_1) holding value r_2; $\mathsf{obj}(r, T)$ means that the heap is exact an entire object of type T, which r refers to. Because the existence of empty objects in OO, we cannot use $r.a \mapsto -$ or $r.a \hookrightarrow -$ to assert the existence of objects in heaps.
- $*$ and $-\!\!*$ are from Separation Logic: $\psi_1 * \psi_2$ means the heap can be split into two parts, where ψ_1 and ψ_2 hold on each part respectively; $\psi_1 -\!\!* \psi_2$ means that if a heap satisfying ψ_1 is added to the heap, the whole heap satisfies ψ_2.

We allow user-defined predicates to extend vocabulary of the assertion language. A predicate definition takes the form $p(\overline{x}) : \psi$, where p is a symbol (predicate name), $\overline{x}$ are its formal parameters, and ψ is the body which is an assertion correlated with $\overline{x}$. Recursive definitions are allowed. In fact, the recursive predicates are indispensable to support specification and verification of programs involving recursive data structures, e.g., lists, trees, etc. Having a definition for symbol p, expression $p(\overline{e})$ can be used as a basic assertion. We use Ψ to denote the set of OOSL assertions.

We use $\psi[v/x]$ (or $\psi[r/x]$, $\psi[r_1/r_2]$) to denote substitutions. We treat $r = v$ the same as $v = r$, and define $v.a \mapsto r$ (which is not a basic assertion here) as $\exists r' \cdot (v = r' \wedge r'.a \mapsto r)$. Some common abbreviations are:

$$r.a \mapsto - \ = \ \exists r' \cdot r.a \mapsto r' \qquad r.a \hookrightarrow r' \ = \ r.a \mapsto r' * \mathbf{true}$$

We use $type(r)$ to denote type of the object which r refers to. Sometimes we need it.

In [19] we defined the semantics for OOSL and proved that most axioms and inference rules for Separation Logic are also correct in OOSL. In practice, we often need to add some mathematical concepts into OOSL, such as relation, set, sequence, etc., to enhance its expressiveness. Such extensions are orthogonal with the core.

We use in this work a small OO language VeriJ which is an extension of a subset of Java with essential OO features. It integrates features of interface, specification and verification with the syntax given in **Fig. 2**, where:

- C and I are class and interface names, respectively. **pub** is used to announce that a date field or predicate is publicly accessible. Mutation, field accessing, casting, method invocation, and object creation are all taken as special assignments.
- We can have a specification π for a constructor or method in a class or interface. In postcondition ψ we can use $\mathbf{old}(e)$ to denote the value of e in the pre-state. Specifications in a supertype can be inherited or overridden in subtypes. If a non-overridden method is not explicitly specified, it takes default "$\langle\mathbf{true}\rangle\langle\mathbf{true}\rangle$".
- User-defined predicates (*specification predicates* in our words) are defined by the form $p(\mathbf{this}, \overline{x}) : \psi$ where the body is ψ, and **this** is written explicitly as the first parameter to denote current object. The non-**pub** predicates in a class are used in method specification to make it abstract and hide implementation details. In addition, methods declared in interfaces have no implementation, and we often need to declare some predicates and specify the methods based on them. When a class C implements an interface I, it should not only provide implementations for methods declared in I, but also definitions for the predicates declared in I, to connect the method specifications in I (and C) with C's implementations.
- L declares an interface which may inherit another interface J. A class C may implement some interfaces $\overline{I}$, and inherit a class B. A sub-interface should not redeclare the same methods of its super-interfaces. As in Java, each class has a superclass, possibly **Object**, but may implement zero or more interfaces. We assume all methods are public. For simple we omit method overloading here. A program G is a (non-empty) sequence of class and interface declarations.

We consider only well-typed programs in verification, and use a static environment to record information in the program text. The environment for program G takes the form $\Gamma_G = (\Delta_G, \Theta_G, \Pi_G, \Phi_G)$, where Δ is the typing environment recording structural information of declarations; Θ is a method lookup environment mapping C, m to its body; Π records method specifications; and Φ records the specification predicates defined in G. We will omit the subscript G when there is no ambiguity.

As said before, an interface defines a type, and a class defines a type with implementation. We always assume that types in discussion are valid in the program, and use C, D for class names, I for interface names, T for type names, to avoid simple conditions. We use $(T, T') \in \Delta.\mathsf{super}$ to mean that T' is a direct supertype of T. We often omit Δ. In the example of **Fig. 1**, we have $\mathsf{super}(D, I_1)$, $\mathsf{super}(D, I_2)$, and $\mathsf{super}(D, B)$. We use $\mathsf{super}(C)$ to get all supertypes of C. In addition, $C <: T$ means that C is a subtype of T ($<:$ is the transitive and reflective closure of super). We use $\Delta(T)$ to get the map from the method name set of T to their signatures, and then $\Delta(T)(m)$ is the signature of method m in type T, with the form of $(\overline{T_1\ z}) : T$. We will use $\Delta(T.m)$ as an abbreviation of $\Delta(T)(m)$. The similar abbreviations are used throughout this paper.

We record all inherited components (fields, method signatures, bodies and specifications, and predicates) for a class as if they were redeclared in the class. Because the basic language supports only single inheritance, all the inherited components are simple to record. A method in a class has a body, either given by a definition, or inherited from the superclass. For a type T, we use $\Theta(T)$ to get the map from method names in T to their bodies. If m is a method name of C, then $\Theta(C.m)$ gets its body. If T is an interface, we suppose $\Theta(T) = \emptyset$. We allow method overridden but not overloading.

For a predicate, we record its publicity (**pub** or not) with the body in Φ, then $\Phi(T.p)$ gives p's definition in T with the **pub** label if existing. For an interface, because there is no implementation, bodies of its predicates are recorded as undef. The specifications are recorded in Π. If method m defined in class C is explicitly specified, its specification is clear; otherwise, m inherits the specification(s) from C's supertypes. If C inherits m from its superclass, it inherits m's specification from the superclass too. Due to the specification inheritance, when C implements interfaces $I_1, I_2, \ldots$, and defines method m without giving a specification, m in C may have multiple specifications $\pi_1, \pi_2, \ldots, \pi_k$ if more than one of the interfaces have specifications for m. In this case, $\Pi(T)$ is a relation from method names to the corresponding specifications. We will use $\{\varphi\}\{\psi\} \in \Pi(T.m)$ in semantic definition to mean that $\{\varphi\}\{\psi\}$ is a specification of m in T, and $\Pi(T.m) = \{\varphi\}\{\psi\}$ when $\{\varphi\}\{\psi\}$ is the only specification.

These components are easy to build by scanning the program text, some details are given in our report [20]. With the environment, the type checking is easy to conduct. We omit it here. One notable fact is that we need also type-checking specifications in VeriJ programs. For a method declaration (or definition) to be well-formed, its pre and post conditions must be well-formed, and the predicates declared in an interface must be realized in its implementation classes, etc. We omit all these details.

4 Verification Framework

Now we develop a framework for modular specification and verification of VeriJ programs. We introduce some notations and definitions first.

We use $\Gamma, C, m \vdash \psi$ to state that assertion ψ holds in method m of class C under Γ. Clearly, here ψ can only be a state-independent assertion. We use $\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\}$ to say that command c in method m satisfies the specification consisting of precondition φ and postcondition ψ. We write $\Gamma \vdash \{\varphi\}\ C.m\{\psi\}$ to state that $C.m$ is correct wrt. specification $\{\varphi\}\{\psi\}$ under Γ, and $\Gamma \vdash \{\varphi\}\ C.C\{\psi\}$ for the constructor C is correct wrt. $\{\varphi\}\{\psi\}$. For a method, because it may have multiple specifications, we use $\Gamma, C.m \vdash \Pi(C.m)$ to say that $C.m$ is correct wrt. its every specification.

For OO programs, behavioral subtyping is crucial in modular verification. To introduce it into our framework, we define a refinement relation between specifications.

Definition 1 (Refinement of Specification). *Given two specifications* $\{\varphi_1\}\{\psi_1\}$ *and* $\{\varphi_2\}\{\psi_2\}$*, we say that the latter one refines the former in context* Γ, C*,* iff *there exists an assertion* R *which is free of the program variables, such that* $\Gamma, C \vdash (\varphi_1 \Rightarrow \varphi_2 * R) \wedge (\psi_2 * R \Rightarrow \psi_1)$*. We use* $\Gamma, C \vdash \{\varphi_1\}\{\psi_1\} \sqsubseteq \{\varphi_2\}\{\psi_2\}$ *to denote this fact.*

For specifications $\{\pi_i\}_i$ *and* $\{\pi'_j\}_j$*, we say* $\{\pi_i\}_i \sqsubseteq \{\pi'_j\}_j$ iff $\forall i\ \exists j \cdot \pi_i \sqsubseteq \pi'_j$. □

Liskov [13] defined the condition for specification refinement as $\varphi_1 \Rightarrow \varphi_2 \wedge \psi_2 \Rightarrow \psi_1$. Above definition is its extension by considering the storage extension and multiple specifications. It follows also the *nature refinement order* proposed in [8].

Interfaces, their inheritance, and the behavioral subtyping relation also correlate with verifications, which we call *static verification*. For an interface I with super-interface I', if method m in I has a new specification $\{\varphi\}\{\psi\}$ overriding its original $\{\varphi'\}\{\psi'\}$ in I', we must verify refinement relation $\Gamma, I \vdash \{\varphi'\}\{\psi'\} \sqsubseteq \{\varphi\}\{\psi\}$. This verification is

$$\text{[H-THIS]}\ \Gamma, T, m \vdash \mathbf{this} : T \qquad \text{[H-SKIP]}\ \Gamma \vdash \{\varphi\}\ \mathbf{skip}\{\varphi\} \qquad \text{[H-ASN]}\ \Gamma \vdash \{\varphi[e/x]\}\ x := e; \{\varphi\}$$

$$\text{[H-MUT]}\ \Gamma \vdash \{v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto \_\}\ v.a := e; \{v = r_1 \wedge e = r_2 \wedge r_1.a \mapsto r_2\}$$

$$\text{[H-LKUP]}\ \Gamma \vdash \{v = r_1 \wedge r_1.a \mapsto r_2\}\ x := v.a; \{x = r_2 \wedge v = r_1 \wedge r_1.a \mapsto r_2\}$$

$$\text{[H-CAST]}\ \Gamma \vdash \{v = r \wedge r <: N\}\ x := (N)v; \{x = r\} \qquad \text{[H-RET]}\ \Gamma \vdash \{\varphi[e/\mathsf{res}]\}\ \mathbf{return}\ e; \{\varphi\}$$

$$\text{[H-SEQ]}\ \frac{\Gamma \vdash \{\varphi\}\ c_1\{\psi\},\ \Gamma \vdash \{\psi\}\ c_2\{R\}}{\Gamma \vdash \{\varphi\}\ c_1\ c_2\{R\}} \qquad \text{[H-COND]}\ \frac{\Gamma \vdash \{b \wedge \varphi\}\ c_1\{\psi\},\ \Gamma \vdash \{\neg b \wedge \varphi\}\ c_2\{\psi\}}{\Gamma \vdash \{\varphi\}\ \mathbf{if}\ b\ c_1\ \mathbf{else}\ c_2\{\psi\}}$$

$$\text{[H-ITER]}\ \frac{\Gamma \vdash \{b \wedge I\}\ c\{I\}}{\Gamma \vdash \{I\}\ \mathbf{while}\ b\ c\{\neg b \wedge I\}} \qquad \text{[H-FRAME]}\ \frac{\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\} \quad \mathsf{FV}(R) \cap \mathsf{MD}(c) = \emptyset}{\Gamma, C, m \vdash \{\varphi * R\}\ c\{\psi * R\}}$$

$$\text{[H-CONS]}\ \frac{\Gamma, C, m \vdash \varphi \Rightarrow \varphi', \quad \Gamma, C \vdash \psi' \Rightarrow \psi \quad \Gamma, C, m \vdash \{\varphi'\}\ c\{\psi'\}}{\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\}} \qquad \text{[H-EX]}\ \frac{\Gamma, C, m \vdash \{\varphi\}\ c\{\psi\} \quad r \text{ is free in } \varphi, \psi}{\Gamma, C, m \vdash \{\exists r \cdot \varphi\}\ c\{\exists r \cdot \psi\}}$$

$$\text{[H-OLD]}\ \frac{\forall \{\varphi\}\{\psi\} \in \Pi(T.m) \bullet \Gamma, T, m \vdash (\overline{z = r} \wedge \varphi[\overline{r}/\overline{z}]) \Rightarrow \psi'}{\Gamma, T, m \vdash \psi'[\mathbf{old}(e)/e]}$$

$$\text{[H-DPRE]}\ \frac{r : D, \quad C <: D, \quad \Phi(D.p(\mathbf{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\mathbf{this}, \overline{a}]} \qquad \text{[H-SPRE]}\ \frac{C <: D, \quad \Phi(D.p(\mathbf{this}, \overline{a})) = \psi}{\Gamma, C, m \vdash D.p(r, \overline{r'}) \Leftrightarrow \mathsf{fix}(D, \psi)[r, \overline{r'}/\mathbf{this}, \overline{a}]}$$

$$\text{[H-PDPRE]}\ \frac{r : D, \quad \Phi(D.p(\mathbf{this}, \overline{a})) = \mathbf{pub}\ \psi}{\Gamma, C, m \vdash p(r, \overline{r'}) \Leftrightarrow \psi[r, \overline{r'}/\mathbf{this}, \overline{a'}]} \qquad \text{[H-PSPRE]}\ \frac{\Phi(D.p(\mathbf{this}, \overline{a})) = \mathbf{pub}\ \psi}{\Gamma, C, m \vdash D.p(r, \overline{r'}) \Leftrightarrow \mathsf{fix}(D, \psi)[r, \overline{r'}/\mathbf{this}, \overline{a}]}$$

Fig. 3. Basic Inference Rules

done only on the logic level, because of no method body in interfaces. This verification is supported by the abstract specifications in our framework.

Now we define the *correct program*, which demands us to verify that every method in a program meets its specification.

Definition 2 (Correct Prog.). *Program G is* correct, iff *for each $\{\varphi\}\{\psi\} \in \Pi_G(T.m)$ (and $\Pi_G(C.C) = \{\varphi\}\{\psi\}$), we have $\Gamma_G \vdash \{\varphi\}T.m\{\psi\}$ (and $\Gamma_G \vdash \{\varphi\}C.C\{\psi\}$).* □

Basic inference rules are given in **Fig. 3**. Rules for assignment, mutation and lookup are similar as their counterparts in Separation Logic. [H-CAST] is special for type casting, and [H-RET] is similar to [H-ASN] but the target is res specially. Rules for composition structures, and rules [H-CONS], [H-EX] take the same forms as what in Hoare logic. [H-FRAME] is the important frame rule, where $\mathsf{FV}(R)$ is the set of all program variables (including internal res) in assertion R, and $\mathsf{MD}(c)$ is the set of variables modified by command c. [H-THIS] is simply a type assertion. [H-OLD] says that if assertion ψ' is provable in the pre-state, then $\psi'[\mathbf{old}(e)/e]$ is provable in the body of the method. A similar rule for constructors is omitted here. Note that here $\forall$ is used only as a shorthand but not a quantifier in logic. Similar notations are used below.

Rules [H-DPRE], [H-SPRE] are key to show our idea that non-public specification predicates have their scopes, and thus can have more than one definitions in the classes crossing the class hierarchy, to implement polymorphism. If a predicate invoked is in scope (in its class or the subclasses), it can be unfolded to its definition. These rules support hiding implementation details, even the details are in the definition of the

predicates used in method specifications. However, these two rules are different. [H-DPRE] says if r is of the type D, then in any subclass of D, $p(r, \overline{r'})$ can be unfolded to the body of p. [H-SPRE] is for the static binding, in that case, $D.p(r, \overline{r'})$ is unfolded to its definition in D, where $\mathsf{fix}(D, \psi)$ gives the *instantiation* of ψ in D:

$$\mathsf{fix}(D,\psi) = \begin{cases} \neg\mathsf{fix}(D,\psi'), & \text{if } \psi \text{ is } \neg\psi' \\ \mathsf{fix}(D,\psi_1) \otimes \mathsf{fix}(D,\psi_2), & \text{if } \psi \text{ is } \psi_1 \otimes \psi_2 \\ \exists r \cdot \mathsf{fix}(D,\psi'), & \text{if } \psi \text{ is } \exists r \cdot \psi' \\ D.q(\mathbf{this}, \overline{r}), & \text{if } \psi \text{ is } q(\mathbf{this}, \overline{r}) \\ \psi, & \text{otherwise.} \end{cases}$$

Here $\otimes$ can be $\vee$, $*$, or $-\!\!*$. Intuitively, fix substitutes predicate names with their definitions in D to their complete names, and then uses the resulting assertion, so it fixes the meaning of an assertion with respect to D. In other words, this function provides a static and fixed explanation for ψ according to a given class. Notice here in unfolding $D.q(r, \overline{r'})$, we use $\mathsf{fix}(D, \psi)$ to fix the meaning of q at first, then do the substitution. With this definition, we can have the correct expansion, and at the same time, avoid infinite expansion in unfolding the recursive defined predicate.

Rules [H-PDPRE] and [H-PSPRE] are similar to [H-DPRE], [H-SPRE], but deal with public predicates. Comparing to above rules, they do not restrict the scope.

Rules related to methods and constructors are given in **Fig. 4** where there is a default side-condition that local variables $\overline{y}$ are not free in φ, ψ. This can be provided by renaming when necessary. An available method in class C can have a specification in C, thus have a definition, or have no specification in C but might be a definition, or an inherited definition with also inherited specification from its superclass. Therefore, we define rules according to these three cases for verifying methods.

[H-MTHD1] is for verifying methods with a specification (and of course a definition) in a class. The rule demands firstly that $C.m$'s body meets its specification, and then asks to check the refinement relations between specifications of m in C and C's supertypes, if existed. Here we promote Π to type set, thus $\Pi(\mathsf{super}(C))(m)$ gives specifications for m in C's supertypes. If there are such specifications, we have to prove the refinement relation with each of them. If there is no, this check is true by default.

[H-MTHD2] is for verifying methods defined in classes without specification. Taking m of C as an example, in this case, we need to verify that the body of m implements correctly with every specification of m in C's supertypes, because C inherits all these specifications, and m may be called from variables of these types. Here $\Pi(C.m)$ is the same set as if we took the type set $\mathsf{super}(C)$ then took all specifications of m from these types. Now we do not need to prove specification refinement relation anymore even if some predicates used in the specifications have been overridden in C. After we have verified m's new body with its each specification in $\Pi(C.m)$, the abstract specifications seem to equivalence from view of the clients.

[H-MINH] is for verifying inherited methods. The rule asks specially to check if m's specification(s) (inheriting from D, maybe more than one because D might inherit some specifications from its supertype(s)) interpreted in C is compatible with its interpretation in D. Here we use $\mathsf{fix}(D, \bullet)$ to fix the meaning of predicates. In addition, we check if the method satisfies each specification of m in C's implementing interfaces (if any) by proving the refinement relation.

$$\text{[H-MTHD1]}\ \frac{\begin{array}{c} C \text{ has specification for } m, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\}, \quad \Pi(C.m) = \{\varphi\}\{\psi\} \\ \Gamma, C, m \vdash \{\mathbf{this} : C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\} \\ \Gamma, C \vdash \Pi(\mathsf{super}(C))(m) \sqsubseteq \{\varphi\}\{\psi\} \end{array}}{\Gamma \vdash \{\varphi\}\ C.m\{\psi\}}$$

$$\text{[H-MTHD2]}\ \frac{\begin{array}{c} C \text{ defines } m \text{ without specification}, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \forall\, \{\varphi\}\{\psi\} \in \Pi(C.m) \bullet \Gamma, C, m \vdash \{\mathbf{this} : C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\} \end{array}}{\Gamma, C.m \vdash \Pi(C.m)}$$

$$\text{[H-MINH]}\ \frac{\begin{array}{c} C \text{ inherits } D.m, \quad \forall\, \{\varphi\}\{\psi\} \in \Pi(C.m) \bullet \Gamma, C \vdash \{\varphi\}\{\psi\} \sqsubseteq \{\mathsf{fix}(D, \varphi)\}\{\mathsf{fix}(D, \psi)\} \\ \forall\, I \in \mathsf{super}(C) \wedge \Pi(I.m) = \{\varphi'\}\{\psi'\} \bullet \Gamma, C \vdash \{\varphi'\}\{\psi'\} \sqsubseteq \Pi(C.m) \end{array}}{\Gamma, C.m \vdash \Pi(C.m)}$$

$$\text{[H-CONSTR]}\ \frac{\begin{array}{c} \Pi(C.C) = \{\varphi\}\{\psi\}, \quad \Theta(C.C) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \Gamma, C, C \vdash \{\overline{z = r} \wedge \overline{y = \mathsf{nil}}\ \wedge \mathsf{raw}(\mathbf{this}, C) * \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\} \end{array}}{\Gamma \vdash \{\varphi\}\ C.C\{\psi\}}$$

$$\text{[H-INV]}\ \frac{\Gamma, C, m \vdash v : T, \quad \{\varphi\}\{\psi\} \in \Pi(T.n)}{\Gamma, C, m \vdash \{v = r \wedge \overline{e = r'} \wedge \varphi[r, \overline{r'}/\mathbf{this}, \overline{z}]\}\ x := v.n(\overline{e})\ \{\psi[r, \overline{r'}, x/\mathbf{this}, \overline{z}, \mathsf{res}]\}}$$

$$\text{[H-NEW]}\ \frac{\Pi(C'.C') = \{\varphi\}\{\psi\}}{\Gamma, C, m \vdash \{\overline{e = r'} \wedge \varphi[\overline{r'}/\overline{z}]\}\ x := \mathbf{new}\ C'(\overline{e})\{\exists r \cdot x = r \wedge \psi[r, \overline{r'}/\mathbf{this}, \overline{z}]\}}$$

Fig. 4. Inference Rules related to Methods and Constructors

Rule [H-CONSTR] is for constructors which has a similar form with [H-MTHD1]. However, a constructor will not have multiple specifications. Here $\mathsf{raw}(\mathbf{this}, C)$ specifies that $\mathbf{this}$ refers to a newly created raw object of type C, and then c modifies its state. The definition of $\mathsf{raw}(r, N)$ is

$$\mathsf{raw}(r, N) \mathrel{\widehat{=}} \begin{cases} \mathsf{obj}(r, N), & N \ \textit{has no field} \\ r : N \wedge (r.a_1 \mapsto \mathsf{nil})\ * \cdots * (r.a_k \mapsto \mathsf{nil}), & \text{fields of } N \text{ is } a_1, \ldots, a_k \end{cases}$$

Last two rules are for method invocation and object creation. Note that $T.n$ may have multiple specifications, and we can use any of them in proving client code. Due to the *behavioral subtyping*, it is enough to do the verification by the type of variable v. Because [H-INV] refers to only specifications, recursive methods are supported.

The soundness of these rules are easy to prove. However, readers may think that [H-MTHD2] is not very satisfactory because it asks for verifying method body for possibly several times. The first answer is that this is necessary, because different specifications for m in the superclass or implemented interfaces may cover different aspects of m's behavior. The definition of m in subclass C must satisfy each of these specifications. However, we may give a new (and weaker) rule to avoid some method body verifications, if we can find that a specification $\{\varphi\}\{\psi\}$ is the strongest:

$$\text{[H-MTHD2']}\ \frac{\begin{array}{c} C \text{ defines } m \text{ without specification}, \quad \Theta(C.m) = \lambda(\overline{z})\{\mathsf{var}\ \overline{y}; c\} \\ \exists\, \{\varphi\}\{\psi\} \in \Pi(C.m) \bullet ((\forall\, \{\varphi'\}\{\psi'\} \in \Pi(C.m) \bullet \Gamma, C \vdash \{\varphi'\}\{\psi'\} \sqsubseteq \{\varphi\}\{\psi\}) \\ \wedge\ \Gamma, C, m \vdash \{\mathbf{this} : C \wedge \overline{z = r} \wedge \overline{y = \mathsf{nil}} \wedge \varphi[\overline{r}/\overline{z}]\}\ c\{\psi[\overline{r}/\overline{z}]\}) \end{array}}{\Gamma, C.m \vdash \Pi(C.m)}$$

Here $\{\varphi\}\{\psi\}$ is the strongest one which refines all the other specifications, including itself. However, if there is no strongest one in $\Pi(C.m)$, this rule will be not applicable, even the method body in C does satisfy all the specifications. In addition, based on the general [H-MTHD2], we may develop some other rules in advance.

Here we see how the information given by developers affects the verification. A given specification for a method is a specific requirement and induces some special proof obligations. It forms a connection between the implementation with the surrounding world: the implemented interfaces, the superclass, and the client codes. When no specification is given, we need to verify more to ensure all the possibilities.

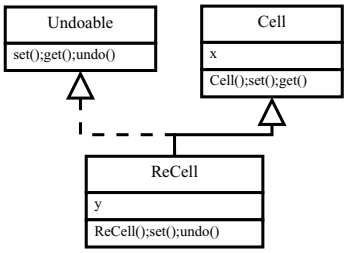
Clearly, our verification framework is modular, because we do not need to re-touch the code in superclass when consider a subclass. In adding a new class to existed code, we just need to verify the new class but not re-consider the existing part. On the other side, in verifying client code, no matter which variable a method invocation is from, be it a variable of a class or an interface, we do not need to consider the real object the variable refers to. This shows that our basic framework, based on the abstract specification and specification predicate concepts, can be naturally extended to support the verification of programs with interfaces. From this extension, we see our framework is nature in dealing with wide spectrum of OO programs. To reveal these good properties, we consider an examples in the next section. More examples can be found in [20].

5 An Example

The hierarchy of classes used here is given in **Fig. 5** using a UML class diagram. It is a variant of a typical example used in OO program's specification and verification study. A class $Cell$ offers simple methods to set and get the value of its field x. We want some classes which can roll back one previous value, and thus declare an interface $Undoable$ with one more method $undo$ than $Cell$. To offer a real class, we define $ReCell$ which inherits $Cell$ and implements $Undoable$. This subclass contains a new field y for saving the old value of x when it is set. To implement the methods declared in $Undoable$, class $ReCell$ defines a method $undo$, inherits $Cell.get$, and overrides $Cell.set$ by a new definition. Here we omit the constructors for simplicity.

Here we give also specifications for methods, in which some specification predicates are used to hide implementation details. Class $Cell$ defines predicate $cell(\mathbf{this}, v)$ to denote that the object holds v, and later in $ReCell$ this predicate is overridden by a new definition. Interface $Undoable$ defines predicates $cell$ and bak to assert the current and backup values, while their implementations are left to the implementing class. Note that predicate $ReCell.cell(\mathbf{this}, v)$ records both current value v in x and some unconcerned value for y, and $ReCell.bak(\mathbf{this}, v)$ records v as the value of y and leaves value of x unconcerned. Because $ReCell.set$ is not explicitly specified, it inherits two specifications from $Undoable.set$ and $Cell.set$. Similarly, $ReCell.get$ inherits a specification from $Cell.get$; and $ReCell.undo$ inherits a specification from $Undoable.undo$. All of these are recorded in the specification environment.

Now we consider how to prove that the program is correct with given specifications by VeriJ inference rules before giving it to clients. Due to the page limited, we only present the main part of the proof process here.



$\mathbf{class}\ \mathit{Cell} : \mathbf{Object}\{$
$\quad \mathbf{Int}\ x;$
$\quad \mathbf{def}\ \mathit{cell}(\mathbf{this}, v) : \mathbf{this}.x \hookrightarrow v;$
$\quad \mathbf{void}\ \mathit{set}(\mathbf{Int}\ v)$
$\quad \langle \mathit{cell}(\mathbf{this}, \_)\rangle\langle \mathit{cell}(\mathbf{this}, v)\rangle$
$\quad \{\ \mathbf{this}.x = v;\ \}$
$\quad \mathbf{Int}\ \mathit{get}()\ \langle \mathit{cell}(\mathbf{this}, v)\rangle$
$\qquad \langle \mathit{cell}(\mathbf{this}, v) \wedge \mathsf{res} = v\rangle$
$\quad \{\ \mathbf{Int}\ c;\ c = \mathbf{this}.x;\ \mathbf{return}\ c;\}$
$\}$

$\mathbf{inter}\ \mathit{Undoable}\{$
$\quad \mathbf{def}\ \mathit{cell}(\mathbf{this}, v);$
$\quad \mathbf{def}\ \mathit{bak}(\mathbf{this}, v);$
$\quad \mathbf{void}\ \mathit{set}(\mathbf{Int}\ v)\ \langle \mathit{cell}(\mathbf{this}, b)\rangle$
$\qquad \langle \mathit{cell}(\mathbf{this}, v) \wedge \mathit{bak}(\mathbf{this}, b)\rangle;$
$\quad \mathbf{Int}\ \mathit{get}()\ \langle \mathit{cell}(\mathbf{this}, v)\rangle$
$\qquad \langle \mathit{cell}(\mathbf{this}, v) \wedge \mathsf{res} = v\rangle;$
$\quad \mathbf{void}\ \mathit{undo}()\ \langle \mathit{bak}(\mathbf{this}, b)\rangle\langle \mathit{cell}(\mathbf{this}, b)\rangle;$
$\}$

$\mathbf{class}\ \mathit{ReCell} : \mathit{Cell} \triangleright \mathit{Undoable}\{$
$\quad \mathbf{Int}\ y;$
$\quad \mathbf{def}\ \mathit{cell}(\mathbf{this}, v) : \mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow \_;$
$\quad \mathbf{def}\ \mathit{bak}(\mathbf{this}, v) : \mathbf{this}.x \hookrightarrow \_ * \mathbf{this}.y \hookrightarrow v;$
$\quad \mathbf{void}\ \mathit{set}(\mathbf{Int}\ v)\ \{\ \mathbf{Int}\ c;$
$\qquad c = \mathbf{this}.x;\ \mathbf{this}.y = c;\ \mathbf{this}.x = v;\}$
$\quad \mathbf{void}\ \mathit{undo}()\ \{\ \mathbf{Int}\ c;$
$\qquad c = \mathbf{this}.y;\ \mathbf{this}.x = c;\}$
$\}$

Fig. 5. Interface *Undoable* and classes *Cell*, *ReCell*

Proving $\mathit{Cell.set}$**:**
$\{\mathit{cell}(\mathbf{this}, \_)\}$
$\{\mathbf{this}.x \hookrightarrow \_\}_{\text{[H-DPRE]}}$
$\mathbf{this}.x = v;$
$\{\mathbf{this}.x \hookrightarrow v\}_{\text{[H-MUT]}}$
$\{\mathit{cell}(\mathbf{this}, v)\}_{\text{[H-DPRE]}}$

Proving $\mathit{Cell.get}$**:**
$\{\mathit{cell}(\mathbf{this}, v) \wedge c = 0\}$
$\{\mathbf{this}.x \hookrightarrow v \wedge c = 0\}$
$c = \mathbf{this}.x;$
$\{\mathbf{this}.x \hookrightarrow v \wedge c = v\}$
$\mathbf{return}\ c;$
$\{\mathit{cell}(\mathbf{this}, v) \wedge \mathsf{res} = v\}$

Proving $\mathit{ReCell.undo}$**:**
$\{\mathit{bak}(\mathbf{this}, b) \wedge c = 0\}$
$\{c = 0 \wedge \mathbf{this}.x \hookrightarrow \_ * \mathbf{this}.y \hookrightarrow b\}$
$c = \mathbf{this}.y;\ \ \mathbf{this}.x = c;$
$\{c = b \wedge \mathbf{this}.x \hookrightarrow c * \mathbf{this}.y \hookrightarrow b\}$
$\{\mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow b\}$
$\{\mathit{cell}(\mathbf{this}, b) \wedge \mathit{bak}(\mathbf{this}, b)\}$
$\{\mathit{cell}(\mathbf{this}, b)\}$

Fig. 6. Proofs for Some Simple Cases

Obviously, to verify methods of *Cell*, we should use rule [H-MTHD1] because all the methods are specified well in *Cell*. Here the premise for specification refinement is vain and thus trivially true. We only need to verify the method bodies. The formal proofs of these two methods are simple and given in **Fig. 6**. As an example, we explain the verifying for the body of $\mathit{Cell.set}$ informally. From its precondition $\mathit{cell}(\mathbf{this}, \_)$, we first apply rule [H-DPRE] to unfold the predicate *cell* used here and get an assertion $\mathbf{this}.x \hookrightarrow \_$, which is the precondition of the next command $\mathbf{this}.x = v$. Then we apply rule [H-MUT] on this command and obtain another assertion $\mathbf{this}.x \hookrightarrow v$ which equals to the postcondition $\mathit{cell}(\mathbf{this}, v)$ according to the rule [H-DPRE] again. Thus, we can conclude that method $\mathit{Cell.set}$ is correct with its specification. The proofs for the other methods are similar, and we omit their explanations to save the space.

For $\mathit{ReCell.set}$ and $\mathit{ReCell.undo}$, we need to prove that their bodies meet the respectively inherited specifications from *ReCell*'s supertypes, because they are not specified explicitly. This asks us to use [H-MTHD2]. The proof for $\mathit{ReCell.undo}$ is simple and given also in **Fig. 6**. For $\mathit{ReCell.set}$, we need to prove that it meets its two inherited

Proving $ReCell.set$ **with** $\Pi(Undoable.set)$**:**	**Proving** $ReCell.set$ **with** $\Pi(Cell.set)$**:**
$\{cell(\mathbf{this}, b) \wedge c = 0\}$	$\{cell(\mathbf{this}, \_) \wedge c = 0\}$
$\{c = 0 \wedge \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow \_\}$	$\{c = 0 \wedge \mathbf{this}.x \hookrightarrow \_ * \mathbf{this}.y \hookrightarrow \_\}$
$c = \mathbf{this}.x;\ \mathbf{this}.y = c;$	$\{c = 0 \wedge \exists b \cdot \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow \_\}$
$\{c = b \wedge \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow c\}$	$c = \mathbf{this}.x;\ \mathbf{this}.y = c;$
$\mathbf{this}.x = v;$	$\{\exists\, b \cdot c = b \wedge \mathbf{this}.x \hookrightarrow b * \mathbf{this}.y \hookrightarrow c\}$
$\{\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow b\}$	$\mathbf{this}.x = v;$
$\{(\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow \_) \wedge$	$\{\exists\, b \cdot \mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow b\}$
$\quad(\mathbf{this}.x \hookrightarrow \_ * \mathbf{this}.y \hookrightarrow b)\}$	$\{\mathbf{this}.x \hookrightarrow v\}$
$\{cell(\mathbf{this}, v) \wedge bak(\mathbf{this}, b)\}$	$\{cell(\mathbf{this}, v)\}$

Fig. 7. Proofs for $ReCell.set$ with two specifications

specifications from $Undoable$ and $Cell$ firstly. The proofs are given in **Fig. 7**. Based on these proofs, we can easily conclude that $ReCell.set$ meets its specification.

In addition, we find that rule [H-MTHD2'] is also applicable here to avoid verifying method body more then one time because there is a refinement relation between the specifications. We show the proof as an example. To work in this way, now we need only check the refinement relation, because we have proved that set meets its specification in $Undoable$. By **Definition 1**, we have trivially:

$$\begin{aligned}\Gamma, ReCell \vdash\ & \{cell(\mathbf{this}, b)\}\{cell(\mathbf{this}, v) \wedge bak(\mathbf{this}, b)\} \Rightarrow \{cell(\mathbf{this}, \_)\}\{cell(\mathbf{this}, v)\} \\ & \Rightarrow \{cell(\mathbf{this}, \_)\}\{cell(\mathbf{this}, v)\} \sqsubseteq \{cell(\mathbf{this}, b)\}\{cell(\mathbf{this}, v) \wedge bak(\mathbf{this}, b)\}\end{aligned}$$

This derivation tells us "$\Gamma, ReCell \vdash \Pi(Cell.set) \sqsubseteq \Pi(Undoable.set)$". Thus, the body of $ReCell.set$ also meets the specification in the superclass $Cell$ according to our weakened rule [H-MTHD2'].

For $ReCell.get$, rule [H-MINH] asks us to prove only specification refinement relations, thus we avoid re-verifying method body and achieve modularity. $ReCell.get$ inherits its specification from $Cell.get$, which is a single specification. Thus, the refinement relation between specifications of $Undoable.get$ and $ReCell.get$ is "$\Gamma, ReCell \vdash \{\varphi'\}\{\psi'\} \sqsubseteq \{\varphi\}\{\psi\}$", where $\varphi = \varphi' = cell(\mathbf{this}, v)$ and $\psi = \psi' = (cell(\mathbf{this}, v) \wedge \mathsf{res} = v)$. This relation holds trivially.

For proving the specification refinement relation between $Cell.get$ and $ReCell.get$, the case is different. Here we must prove

$$\begin{aligned}\Gamma, ReCell \vdash\ & \{cell(\mathbf{this}, v)\}\{cell(\mathbf{this}, v) \wedge \mathsf{res} = v\} \\ & \sqsubseteq \{\mathsf{fix}(Cell, cell(\mathbf{this}, v))\}\{\mathsf{fix}(Cell, cell(\mathbf{this}, v) \wedge \mathsf{res} = v)\}\end{aligned}$$

By the **Definition 1** and the definition of fix, we need to prove that there exists an assertion R such that

$$\begin{aligned}\Gamma, ReCell \vdash\ & (cell(\mathbf{this}, v) \Rightarrow \mathsf{fix}(Cell, cell(\mathbf{this}, v)) * R) \wedge \\ & \quad (\mathsf{fix}(Cell, cell(\mathbf{this}, v) \wedge \mathsf{res} = v) * R \Rightarrow (cell(\mathbf{this}, v) \wedge \mathsf{res} = v)) \\ & \Rightarrow (cell(\mathbf{this}, v) \Rightarrow Cell.cell(\mathbf{this}, v) * R) \wedge \\ & \quad ((Cell.cell(\mathbf{this}, v) \wedge \mathsf{res} = v) * R \Rightarrow (cell(\mathbf{this}, v) \wedge \mathsf{res} = v)) \\ & \Rightarrow ((\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow \_) \Rightarrow (\mathbf{this}.x \hookrightarrow v * R)) \wedge \\ & \quad (((\mathbf{this}.x \hookrightarrow v * R) \wedge \mathsf{res} = v) \Rightarrow ((\mathbf{this}.x \hookrightarrow v * \mathbf{this}.y \hookrightarrow \_) \wedge \mathsf{res} = v))\end{aligned}$$

Note that, $\mathsf{res} = v$ is a pure, so when inferring from the second assertion to the third one, we apply an axiom in OOSL, which says that, if an assertion q is pure, we have

```
Bool cell_test()
⟨true⟩⟨res = rtrue⟩
{
  Int c1, c2;
  Bool b = false;
  Cell t1;
  Undoable t2;
  t1 = new ReCell();
  t2 = (Undoable)t1;
  t1.set(5);
  c1 = t2.get();
  t2.set(3);
  t2.undo();
  c2 = t1.get();
  if (c1==c2) b = true;
  return b;
}
```

```
{c1 = 0 ∧ c2 = 0 ∧ b = rfalse ∧ t1 = rnull ∧ t2 = rnull}
t1 = new ReCell(); t2 = (Undoable)t1;
{∃r1, r2, v1, v2 · t1 = r1 ∧ t2 = r2 ∧ c1 = 0 ∧ c2 = 0 ∧
   b = rfalse ∧ r1 = r2 ∧ cell(r1, v1) ∧ bak(r1, v2)}
t1.set(5); c1 = t2.get();
{∃r1, r2, v1 · t1 = r1 ∧ t2 = r2 ∧ c1 = 5 ∧ c2 = 0 ∧
   b = rfalse ∧ r1 = r2 ∧ cell(r1, 5) ∧ bak(r1, v1)}
t2.set(3);
{∃r1, r2 · t1 = r1 ∧ t2 = r2 ∧ c1 = 5 ∧ c2 = 0 ∧ b = rfalse ∧
   r1 = r2 ∧ cell(r1, 3) ∧ bak(r1, 5)}
t2.undo(); c2 = t1.get();
{∃r1, r2 · t1 = r1 ∧ t2 = r2 ∧ c1 = 5 ∧ c2 = 5 ∧ b = rfalse∧
   r1 = r2 ∧ cell(r1, 5) ∧ bak(r1, 5)}
if (c1==c2)
  {∃r1, r2 · t1 = r1 ∧ t2 = r2 ∧ c1 = 5 ∧ c2 = 5 ∧ b = rfalse ∧
     c1 = c2 ∧ r1 = r2 ∧ cell(r1, 5) ∧ bak(r1, 5)}
  b = true;
  {∃r1, r2 · t1 = r1 ∧ t2 = r2 ∧ c1 = 5 ∧ c2 = 5 ∧ b = rtrue ∧
     c1 = c2 ∧ r1 = r2 ∧ cell(r1, 5) ∧ bak(r1, 5)}
{b = rtrue}
return b;
{res = rtrue}
```

Fig. 8. A Client Method and Its Proof

$(p \wedge q) * r \Rightarrow (p * r) \wedge q$. Let "$R = \mathbf{this}.y \hookrightarrow -$", then we have the above implications true easily. Therefore, we can conclude that $ReCell.get$ meets its specification.

Now we show how a client can be verified by just referring to the specifications in interfaces and classes, thus is done abstractly and modularly. In **Fig. 8** (left), we define a method $cell\_test$ which declares a variable of type $Cell$ but actually assigns it an object of $ReCell$. Then a new variable t_2 is declared and assigned the same object by casting t_1 to $Undoable$. We give the proof of this method in detail in the figure too. The proof involves only the abstract specifications of the interface and classes.

6 Related Work and Conclusion

To support specifications and verification of OO programs with interface types, we develop here a formal framework which offers modularity for both specification and verification. The OO language VeriJ used here takes the pure reference semantics. A version of Separation Logic, named OOSL, is used for specifying and reasoning VeriJ programs. We suggest *abstract specifications* for describing behaviors of methods. This technique can support also "behavioral" specification for the method declarations in interfaces which have no implementations. We introduce *specification predicates* to link abstract specifications with implementation details, which serve also the connection between the classes with the interfaces which they implement.

We design rules for visibility, inheritance and overriding of specification predicates and method specifications, and develop a set of inference rules which can derive proof

obligations from program with specifications for verifying VeriJ programs. Our approach supports full encapsulation for the implementation details, and can also avoid re-verification of inherited methods. In contrast to the work presented in [15, 4], we use only one specification for each method. As in the main-stream OO languages, e.g. Java and C#, here one class may implement several interfaces, as well as inherit a superclass. Our framework supports inheriting multi-specifications for methods from implemented interfaces and the superclass. We define the refinement relation for multi-specifications, and propose inference rules for proving programs in this situation. By an example, with more examples in our report [20], we show that the framework can deal with various common problems encountered in OO practice.

The research on the specification and verification in JML and Spec# frameworks [9, 7, 10, 2, 1] considered also interface types. Similarly, these frameworks support method specifications in interfaces, and allow specification inheritance. The refinement relations between supertypes and their subtypes are defined to pursue modular reasoning and behavioral subtyping. Differently, Spec# requires overriding methods in a subtype inherit the same preconditions from its supertypes while postconditions can be strengthened. This brings a big constraint on implementations in subtypes. Actually, to allow more flexible behaviors, we should permit not only strengthening postconditions but also weakening preconditions in subtypes, as what we and JML do. Notably, our framework is more general. In one side, we develop an approach for the inheritance with multiple specifications for methods. In addition, our definition for specification refinement allows storage extension of subclass, which is necessary for dealing with mutable OO structures but totally omitted in Spec# and JML frameworks.

Abstraction techniques adopted in JML and Spec# are similar. Both use model fields and calls of pure methods in their specifications. We find such calls in specifications are not abstract and convenient enough for clients to use and understand, because it may enforce clients to know what these pure methods do. We propose specification predicates to hide information from clients and use them in specifications in interfaces (and classes) to provide enough information for verifying class and client codes conveniently. In addition, as pointed by [15], the early work, including JML and Spec#, can not avoid re-verification of the inherited methods. That might be another weakness of the approaches based on the model fields, pure methods, etc.

In this work, we utilize structures in programs and specification predicates as the semantic link over the class hierarchy, rather than linking the *abstract predicate families* to classes by the type of their first parameter and a tag [15]. Using one specification for a method, we can get rid of repeated expressions, and express the semantic decision for the class only in the local defined predicates. This feature makes it better to support the *single point rule* in the specifications. In addition, it is not clear how the abstract predicate families and dual specifications mechanisms can be used (extended) to support specification and verification of OO programs with interface types.

By successfully extending our framework to support the interface features, we see more clearly the usefulness of the concept specification predicates and its potential power. In fact, the key point of our approach is to introduce polymorphism concepts into the specification and verification framework that is learnt from the successful OO practice. As the future work, we will explore further the potentials of our approach, to

support more OO and verification features, such as object invariants, frame problems and confinement, open programs, and so on.

References

1. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: the spec# experience. Communications of the ACM 54(6), 81–91 (2011)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract. Software: Practice and Experience 35(6), 583–599 (2005)
4. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular OO verification with separation logic. In: POPL 2008, pp. 87–99. ACM, New York (2008)
5. Distefano, D., Parkinson, M.J.: jstar: Towards practical verification for java. ACM SIGPLAN Notices 43(10), 213–226 (2008)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addlison Wesley (1994)
7. Leavens, G.T.: JML's Rich, Inherited Specifications for Behavioral Subtypes. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 2–34. Springer, Heidelberg (2006)
8. Leavens, G.T., Naumann, D.A.: Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, Ames, Iowa, 50011 (2006)
9. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31(3), 1–38 (2006)
10. Leavens, G.T., Müller, P.: Information hiding and visibility in interface specifications. In: 29th International Conference on Software Engineering, ICSE 2007, pp. 385–395 (2007)
11. Leino, K.R.M.: Toward reliable modular programs. PhD thesis, California Institute of Technology, Pasadena, CA, USA, UMI Order No. GAX95-26835 (1995)
12. Leino, K.R.M.: Data groups: specifying the modification of extended state. SIGPLAN Notices 33, 144–153 (1998)
13. Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Transactions on Programing Languages and Systems 16(6), 1811–1841 (1994)
14. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
15. Parkinson, M.J., Bierman, G.M.: Separation logic, abstraction and inheritance. In: POPL 2008, pp. 75–86. ACM, New York (2008)
16. Poetzsch-Heffter, A.: Specification and verification of object-oriented programs. Technische Universität München (1997)
17. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
18. Yijing, L., Ali, H., Zongyan, Q.: Inheritance and modularity in specification and verification of OO programs. In: TASE 2011, pp. 19–26. IEEE Computer Society (2011)
19. Yijing, L., Zongyan, Q.: A Separation Logic for OO Programs. In: Barbosa, L.S., Lumpe, M. (eds.) FACS 2010. LNCS, vol. 6921, pp. 88–105. Springer, Heidelberg (2010)
20. Zongyan, Q., Ali, H., Yijing, L.: Modular verification of OO programs with interface types. Technical report, School of Math., Peking Univ. (2012), http://www.mathinst.pku.edu.cn/download.php?classid=22

Separation Predicates: A Taste of Separation Logic in First-Order Logic*

François Bobot and Jean-Christophe Filliâtre

LRI, Univ Paris-Sud, CNRS, Orsay F-91405
INRIA Saclay-Île-de-France, ProVal, Orsay F-91893

Abstract. This paper introduces *separation predicates*, a technique to reuse some ideas from separation logic in the framework of program verification using a traditional first-order logic. The purpose is to benefit from existing specification languages, verification condition generators, and automated theorem provers. Separation predicates are automatically derived from user-defined inductive predicates. We illustrate this idea on a non-trivial case study, namely the composite pattern, which is specified in C/ACSL and verified in a fully automatic way using SMT solvers Alt-Ergo, CVC3, and Z3.

1 Introduction

Program verification has recently entered a new era. It is now possible to prove rather complex programs in a reasonable amount of time, as demonstrated in recent program verification competitions [17,12,10]. One of the reasons for this is tremendous progress in automated theorem provers. SMT solvers, in particular, are tools of choice to discharge verification conditions, for they combine full first-order logic with equality, arithmetic, and a handful of other theories relevant to program verification, such as arrays, bit vectors, or tuples. Notable examples of SMT solvers include Alt-Ergo [4], CVC3 [1], Yices [9], and Z3 [8].

Yet, when it comes to verifying programs involving pointer-based data structures, such as linked lists, trees, or graphs, the use of traditional first-order logic to specify, and of SMT solvers to verify, shows some limitations. Separation logic [22] is then an elegant alternative. Designed at the turn of the century, it is a program logic with a new notion of conjunction to express spatial separation. Separation logic requires dedicated theorem provers, implemented in tools such as Smallfoot [2] or VeriFast [13,15]. One drawback of such provers, however, is to either limit the expressiveness of formulas (*e.g.* to the so-called symbolic heaps), or to require some user-guidance (*e.g.* open/close commands in Verifast).

In an attempt to conciliate both approaches, we introduce the notion of *separation predicates*. The idea is to introduce some ideas from separation logic into a traditional verification framework where the specification language, the

* This work was (partially) supported by the Information and Communication Technologies (ICT) Programme as Project FP7-ICT-2009-C-243881 CerCo and by the U3CAT project (ANR-08-SEGI-021) of the French national research organization.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 167–181, 2012.

verification condition generator, and the theorem provers were not designed with separation logic in mind. Separation predicates are automatically derived from user-defined inductive predicates, on demand. Then they can be used in program annotations, exactly as other predicates, *i.e.*, without any constraint. Simply speaking, where one would write $P \star Q$ in separation logic, one will here ask for the generation of a separation predicate *sep* and then use it as $P \wedge Q \wedge sep(P, Q)$.

We have implemented separation predicates within Frama-C's plug-in Jessie for deductive verification [21]. This paper demonstrates the usefulness of separation predicates on a realistic, non-trivial case study, namely the composite pattern from the VACID-0 benchmark [20]. We achieve a fully automatic proof using three existing SMT solvers.

This paper is organized as follows. Section 2 gives a quick overview of what separation predicates are, using the classic example of list reversal. Section 3 formalizes the notion of separation predicates and briefly describes our implementation. Then, Section 4 goes through the composite pattern case study. Section 5 presents how this framework can be extended to express the set of pointers modified by a function. We conclude with related work in Section 6.

2 Motivating Example

As an example, let us consider the classic in-place list reversal algorithm:

$rev(p) \equiv$
  $q := \texttt{NULL}$
  while $p \neq \texttt{NULL}$ do $t := p{\rightarrow}\texttt{next}$; $p{\rightarrow}\texttt{next} := q$; $q := p$; $p := t$ done
  return q

We may want to verify that, whenever p points to a finite singly-linked list, then $rev(p)$ returns a finite list. (Proving that lists are indeed reversed requires more space than available here.) To do so, we first define the notion of finite singly-linked lists, for instance using the following inductive predicate *islist*:

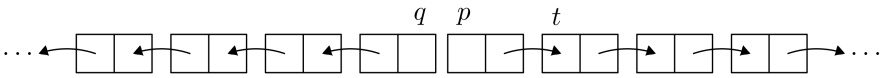
$$\begin{array}{l}\textsf{inductive}\ islist(p)\ \equiv \\ \quad \mid C_0 : islist(\texttt{NULL}) \\ \quad \mid C_1 : \forall p.\, p \neq \texttt{NULL} \Rightarrow islist(p{\rightarrow}\texttt{next}) \Rightarrow islist(p)\end{array}$$

Then we specify function *rev* using the following Hoare triple:

$$\{islist(p)\}\ q := rev(p)\ \{islist(q)\}$$

To perform the proof, we need a loop invariant. A natural invariant expresses that both p and q are finite lists, that is $islist(p) \wedge islist(q)$.

Unfortunately, this is not enough for the proof to be carried out. Indeed, we lack the crucial information that assigning $p{\rightarrow}\texttt{next}$ will not modify lists q and t. Therefore, we cannot prove that the invariant above is preserved.



Separation logic proposes an elegant solution to this problem. It introduces a new logical connective $P \star Q$ that acts as the conjunction $P \wedge Q$ and expresses spatial separation of P and Q at the same time. In the list reversal example, it is used at two places. First, it is used in the definition of *islist* to express that the first node of a list is disjoint from the remaining nodes:

$$islist(p) \;\equiv\; \mathsf{if}\ p = \texttt{NULL}\ \mathsf{then}\ \mathsf{emp}\ \mathsf{else}\ \exists q.\, p{\rightarrow}\texttt{next} \mapsto q \star islist(q)$$

This way, we can now prove that list t is preserved when $p{\rightarrow}\texttt{next}$ is assigned. Second, the connective $\star$ is also used in the loop invariant to express that lists p and q do not share any pointer:

$$islist(p) \star islist(q).$$

This way, we can now prove that list q is preserved when $p{\rightarrow}\texttt{next}$ is assigned. Using a dedicated prover for separation logic, list reversal can be proved correct using this loop invariant.

In our attempt to use traditional SMT solvers instead, we introduce the notion of *separation predicates*: the $\star$ connective of separation logic is replaced by new predicate symbols, which are generated on a user-demand basis. Our annotated C code for list reversal using separation predicates is given in Fig. 1.

We define predicate `islist` inductively (lines 4–8), as we did earlier in this section. In this definition `\valid(p)` express that `p` is a pointer that can be safely dereferenced (allocated and not freed). It captures finite lists only and, consequently, the first node of a list is disjoint from the remaining nodes. However, such a proof requires induction and thus is out of reach of SMT solvers. We add this property as a lemma (lines 11–12), using a separation predicate `sep_node_islist` (introduced at line 10). This lemma is analogous to the $\star$ used in the definition of `islist` in separation logic. To account for the $\star$ in the loop invariant, we first introduce a new separation predicate `sep_islist_islist` (line 14) and then we use it in the loop invariant (line 21).

With these annotations, the axiomatizations and the definitions automatically generated for `sep_node_islist` and `sep_islist_islist` allow a general-purpose SMT solver such as Alt-Ergo or CVC3 to discharge all verification conditions obtained by weakest precondition for the code in Fig. 1, in no time.

3 Separation Predicates

3.1 Inductive Definitions

A separation predicate is generated from user-defined inductive predicates. The generation is sound only if the definitions of the inductive predicates obey several constraints, the main one being that two distinct cases should not overlap. Fortunately, this is the case for most common inductive predicates. For instance, predicate `islist` from Fig. 1 (lines 4–8) trivially satisfies the non-overlapping constraint, since `p` cannot be both null and non-null.

Generally speaking, we consider inductive definitions following the syntax given in Fig. 2. The constraints are then the following:

```
1  struct node { int hd; struct node *next; };
2
3  /*@
4  inductive islist(struct node *p) {
5   case nil: islist(\null);
6   case cons: \forall struct node *p; p != \null ==> \valid(p) ==>
7   islist(p->next) ==> islist(p);
8  }
9
10 #Gen_Separation sep_node_islist(struct node*, islist)
11 lemma list_sep:
12   \forall struct node *p; p!=null ==>
13     islist(p) ==> sep_node_islist(p, p->next);
14
15 #Gen_Separation sep_islist_islist(islist, islist)
16 @*/
17
18 /*@ requires islist(p); ensures islist(\result); @*/
19 struct node * rev(struct node *p) {
20   struct node *q = NULL;
21   /*@ loop invariant
22         islist(p) && islist(q) && sep_islist_islist(p,q); @*/
23   while(p != NULL) {
24     struct node *tmp = p->next;
25     p->next = q;
26     q = p;
27     p = tmp;
28   }
29   return q;
30 }
```

Fig. 1. List Reversal

$$
\begin{array}{rrl}
\text{(terms)} & t & ::= \quad x \mid t{\rightarrow}\texttt{field} \mid \phi(\boldsymbol{t}) \\
\text{(formulas)} & f & ::= \quad t = t \mid \neg(t = t) \mid p(\boldsymbol{x}) \\
\text{(inductive case)} & c & ::= \quad \texttt{C} : \forall \boldsymbol{x}. f \Rightarrow \ldots \Rightarrow f \Rightarrow p(\boldsymbol{x}) \\
\text{(inductive definition)} & d & ::= \quad \texttt{inductive}\, p(\boldsymbol{x}) = c \mid \ldots \mid c
\end{array}
$$

Fig. 2. Inductive Definitions

- in a term t, a function symbol ϕ cannot refer to the memory state;
- in a formula f, a predicate symbol p can refer to the memory state only if it is an inductively defined predicate following the constraints (which includes the predicate being defined);
- if $\texttt{C}_\texttt{i} : \forall \boldsymbol{x}. f_{i,1} \Rightarrow \ldots \Rightarrow f_{i,n_i} \Rightarrow p(\boldsymbol{x})$ and $\texttt{C}_\texttt{j} : \forall \boldsymbol{x}. f_{j,1} \Rightarrow \ldots \Rightarrow f_{j,n_j} \Rightarrow p(\boldsymbol{x})$ are two distinct cases of $\texttt{inductive}\, p(\boldsymbol{x}) = c_1 \mid \ldots \mid c_n$, then we should have

$$\forall \boldsymbol{x}. \neg(f_{i,1} \wedge \cdots \wedge f_{i,n_i} \wedge f_{j,1} \wedge \cdots \wedge f_{j,n_j}).$$

It is worth pointing out that an inductive predicate which is never used to define a separation predicate does not have to follow these restrictions.

3.2 An Axiomatization of Footprints

The footprint of an inductive predicate p is the set of pointers which it depends on. More precisely, in a memory state m where $p(\boldsymbol{x})$ is true, the pointer q is in the footprint of $p(\boldsymbol{x})$ if we can modify the value q points at such that $p(\boldsymbol{x})$ does not hold anymore. Such a definition is too precise to be used in practice. We use instead a coarser notion of footprint, which is derived from the definition of p and over-approximates the precise footprint.

Let us consider the definition of `islist`. First, we introduce a new type `ft` for footprints. Then we declare a function symbol $\texttt{ft}_{\texttt{islist}}$ and a predicate symbol $\in$. The intended semantics is the following: $\texttt{ft}_{\texttt{islist}}(m, p)$ is the footprint of $\texttt{islist}(p)$ in memory state m and $q \in \texttt{ft}_{\texttt{islist}}(m, p)$ means that q belongs to the footprint $\texttt{ft}_{\texttt{islist}}(m, p)$. Both symbols are axiomatized simultaneously as follows:

$$\forall q. \forall m. \forall p.\, q \in \texttt{ft}_{\texttt{islist}}(m, p) \Leftrightarrow \begin{pmatrix} p \neq \texttt{NULL} \wedge \texttt{islist}(m, \{p{\rightarrow}\texttt{next}\}_m) \\ \wedge (q = p \vee q \in \texttt{ft}_{\texttt{islist}}(m, \{p{\rightarrow}\texttt{next}\}_m)) \end{pmatrix}$$

where $\{p{\rightarrow}\texttt{next}\}_m$ stands for expression $p{\rightarrow}\texttt{next}$ in memory state m.

Then separation predicates are easily defined from footprints. The pragma from line 10 in Fig. 1 generates the definition

$$\texttt{sep\_node\_islist}(m, q, p) \triangleq q \notin \texttt{ft}_{\texttt{islist}}(m, p)$$

and pragma from line 14 generates the definition

$$\begin{aligned} &\texttt{sep\_islist\_islist}(m, p_1, p_2) \triangleq \\ &\quad \forall q.\, q \notin \texttt{ft}_{\texttt{islist}}(m, p_1) \vee q \notin \texttt{ft}_{\texttt{islist}}(m, p_2) \end{aligned}$$

(where $q \notin s$ stands for $\neg(q \in s)$). The predicate symbols and the types that appears in the pragma specify the signature of the separation predicate and which inductive predicate must be used to defined the separation predicate. A type is viewed as the predicate symbol of an unary predicate of this type whose footprint is reduced to its argument. The signature of the defined separation predicate is the concatenation of the signature of the predicate symbols.

Generally speaking, in order to axiomatize the footprint of an inductive predicate, we first introduce a meta-operation $\texttt{FT}_{m,q}(e)$ that builds a formula expressing that q is in the footprint of a given expression e in memory state m:

$$\begin{aligned} \texttt{FT}_{m,q}(x) &= \bot \\ \texttt{FT}_{m,q}(t{\rightarrow}\texttt{j}) &= \texttt{FT}_{m,q}(t) \vee q = t \\ \texttt{FT}_{m,q}(\phi(\boldsymbol{t})) &= \bigvee_j \texttt{FT}_{m,q}(t_j) \\ \texttt{FT}_{m,q}(t_1 = t_2) &= \texttt{FT}_{m,q}(\neg(t_1 = t_2)) = \texttt{FT}_{m,q}(t_1) \vee \texttt{FT}_{m,q}(t_2) \\ \texttt{FT}_{m,q}(p(\boldsymbol{t})) &= \bigvee_j \texttt{FT}_{m,q}(t_j) \vee q \in \texttt{ft}_p(m, \boldsymbol{t}) \end{aligned}$$

We pose $q \in \mathtt{ft}_p(m, \boldsymbol{t}) \triangleq \bot$ whenever predicate p does not depend on the memory state. Then the footprint of an inductive predicate p defined by $\mathtt{inductive}\, p(\boldsymbol{x}) = c_1 | \ldots | c_n$ with c_i being $\mathtt{C_i} : \forall \boldsymbol{x}. f_{i,1} \Rightarrow \ldots \Rightarrow f_{i,n_i} \Rightarrow p(\boldsymbol{x})$ is axiomatized as follows:

$$\forall q. \forall m. \forall \mathbf{x}.\, q \in \mathtt{ft}_p(m, \mathbf{x}) \Leftrightarrow \bigvee_i \left(\bigwedge_j \overline{f_{i,j}} \wedge \bigvee_j \mathtt{FT}_{m,q}(f_{i,j}) \right)$$

where $\overline{f_{i,j}}$ is the version of $f_{i,j}$ with the memory explicited (eg. $\overline{t \rightarrow \mathtt{j}} = \{t \rightarrow \mathtt{j}\}_m$). In the axiom above for the footprint of `islist`, we simplified the `NULL` case since it is equivalent to $\bot$.

With the footprints of the inductive predicates you can now define the separation predicate. A separation predicate that define the separation of n inductive predicates is defined as the conjunction of all the disjunction $q \in \mathtt{ft}_{p_i}(m, \boldsymbol{x_i}) \vee q \in \mathtt{ft}_{p_j}(m, \boldsymbol{x_j})$ between the footprint of the inductive predicate. The soundness of this construction have been proved in [3].

The separation predicates allow you to translate a large set of separation logic formulas, namely first-order separation logic formula without magic wand and with separation conjunction used only on inductive predicates which definitions satisfy our constraints.

3.3 Mutation Axioms

The last ingredient we generate is a mutation axiom. It states the main property of the footprint, namely that an assignment outside the footprint does not invalidate the corresponding predicate. In the case of `islist`, the mutation axiom is

$$\forall m, p, q, v.\, q \notin \mathtt{ft}_{\mathtt{islist}}(m, p) \Rightarrow \mathtt{islist}(m, p) \Rightarrow \mathtt{islist}(m[q \rightarrow \mathtt{next} := v], p)$$

where $m[q \rightarrow \mathtt{next} := v]$ stands for a new memory state obtained from m by assigning value v to memory location $q \rightarrow \mathtt{next}$. Actually, this property could be proved from the definition of $\mathtt{ft}_{\mathtt{islist}}$, but this would require induction. Since this is out of reach of SMT solvers, we state it as an axiom. We do not require the user to discharge it as a lemma, since it is proved sound in the meta-theory [3]. This is somehow analogous to the mutation rule of separation logic, which is proved sound in the meta-theory. The mutation rule of separation logic also allows proving that two formulas stay separated if you modify something separated from both of them. We can prove the same by adding an autoframe axiom, which is reminiscent of the autoframe concept in dynamic frames [16]:

$$\begin{aligned} \forall m, p, q, v.\, & q \notin \mathtt{ft}_{\mathtt{islist}}(m, p) \Rightarrow \mathtt{islist}(m, p) \Rightarrow \\ & \mathtt{ft}_{\mathtt{islist}}(m, p) = \mathtt{ft}_{\mathtt{islist}}(m[q \rightarrow \mathtt{next} := v], p) \end{aligned}$$

Generally speaking, for each inductive predicate p and for each field `field` we add the following axioms :

$$\forall q. \forall v. \forall m. \forall \boldsymbol{x}. \neg q \in \mathtt{ft}_p(m, \boldsymbol{x}) \Rightarrow p(m, \boldsymbol{x}) \Rightarrow p(m[q \rightarrow \mathtt{field} := v], \boldsymbol{x})$$

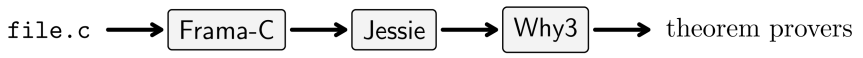
and

$$\forall q.\forall v.\forall m.\forall \boldsymbol{x}.\ \neg q \in \mathtt{ft}^c_p(m,\boldsymbol{x}) \Rightarrow p(m,\boldsymbol{x}) \Rightarrow \\ \mathtt{ft}_p(m,\mathtt{x}) = \mathtt{ft}_p(m[q{\rightarrow}\mathtt{field} := v],\boldsymbol{x}).$$

The distinctness of the cases of the inductive predicate p appears in the proof of the autoframe property.

3.4 Implementation

Our generation of separation predicates is implemented in the Frama-C/Jessie tool chain for the verification of C programs [11,21,5]. This tool chain can be depicted as follows:



From a technical point of view, our implementation is located in the Jessie tool, since this is the first place where the memory model is made explicit[1]. Jessie uses the component-as-array model also known as the Burstall-Bornat memory model [7,6]. Each structure field is modeled using a distinct applicative array. Consequently, function and predicate symbols such as $\mathtt{ft}_{\mathtt{islist}}$ or `islist` do not take a single argument m to denote memory state, but one or several applicative arrays instead, one for each field mentioned in the inductive definition. Similarly, a quantification $\forall m$ in our meta-theory (Sec. 3.2 and 3.3 above) is materialized in the implementation by one or several quantifications over applicative arrays, one for each field appearing in the formula. In the case of `islist`, for instance, quantification $\forall m$ becomes $\forall \mathtt{next}$, expression $\{p{\rightarrow}\mathtt{next}\}_m$ becomes $\mathtt{get}(\mathtt{next},p)$, and expression $m[q{\rightarrow}\mathtt{next} := v]$ becomes $\mathtt{set}(\mathtt{next},p,v)$, where `get` and `set` are access and update operations over applicative arrays. Additionally, we have to define one footprint symbol for each field.

It is worth pointing out that we made no modification at all in Why3 to support our separation predicates. Only Jessie has been modified.

4 A Case Study: Composite Pattern

To show the usefulness of separation predicates, we consider the problem of verifying an instance of the *Composite Pattern*, as proposed in the VACID-0 benchmark [20].

4.1 The Problem

We consider a forest, whose nodes are linked upward through parent links. Each node carries an integer value, as well as the sum of the values for all nodes in its subtree (including itself). The corresponding C structure is thus defined as follows:

[1] Since we could not extend the ACSL language with the new pragmas for separation, we have to modify the Jessie input file manually at each run. Furthermore we use in the assigns clauses the keyword `\all` that does not exist yet in ACSL.

```
struct node {
  int val, sum;
  struct node *parent;
};
typedef struct node *NODE;
```

The operations considered here are the following: `NODE create(int v);`, creates a new node; `void update(NODE p, int v);`, assigns value `v` to node `p`; `void addChild(NODE p, NODE q);`, set node `p` as `q`'s parent, assuming node `q` has no parent; `void dislodge(NODE p);`, disconnects `p` from its parent, if any.

One challenge with such a data structure is that operations `update`, `addChild`, and `dislodge` have non-local consequences, as the `sum` field must be updated for all ancestors. Another challenge is to prevent `addChild` from creating a cycle, *i.e.*, to express that node `q` is not already an ancestor of node `p`. Thus we prove the memory safety and the correct behavior of these operations.

4.2 Code and Specification

Our annotated C code for this instance of the composite pattern is given in the appendix. In this section, we comment on the key aspects of our solution. The annotations are written in the ACSL specification language. The behavior of the functions are defined by contract: the keyword `requires` introduces the precondition expressed by a first-order formula, the keyword `ensures` introduces the post-conditions, and the keyword `assigns` introduces the set of memory location that can be modified by a call to the function. The precondition and this set are interpreted before the execution of the function, the post-conditions is interpreted after. One can refer in the post-condition to the state before the execution of the function using the keyword `\old`. It must be remarked that if a field of a type is never modified in the body of a function you don't need to mention it in the assigns clauses. Moreover the component-as-array memory model ensures without reasoning that any formulas that depend only of such fields remain true after a call to the function.

Separation Predicate. For the purpose of `addChild`'s specification, we use a separation predicate. It states that a given node is disjoint from the list of ancestors of another node. Such a list is defined using predicate `parents` (lines 7–12), which is similar to predicate `islist` in the previous section. The separation predicate, `sep_node_parents`, is then introduced on line 14 and used in the precondition of `addChild` on line 84.

This is a crucial step, since otherwise assignment `q->parent = p` on line 95 could break property `parents(p)`. Such a property is indeed required by `upd_inv` to ensure its termination.

Restoring the Invariant. As suggested in VACID-0 [20], we introduce a function to restore the invariant (function `upd_inv` on lines 68–77). Given a node `p` and an offset `delta`, it adds `delta` to the `sum` field of `p` and of all its ancestors.

This way, we reuse this function in `addChild` (with the new child's sum), in `update` (with the difference), and in `dislodge` (with the opposite of the child's sum).

Local and Global Invariant. Another key ingredient of the proof is to ensure the invariant property that, for each node, the `sum` field contains the sum of values in all nodes beneath, including itself. To state such a property, we need to access children nodes. Since the data structure does not provide any way to do that (we only have parent links), we augment the data structure with ghost children links. To make it simple, we assume that each node has at most two children, stored in ghost fields `left` and `right` (line 4). Structural invariants relating fields `parent`, `left`, and `right` are gathered in predicate `wf` (lines 28–37).

To state the invariant for `sum` fields, we first introduce a predicate `good` (lines 20–23). It states that the `sum` field of a given node `p` has a correct value when `delta` is added to it. It is important to notice that predicate `good` is a *local* invariant, which assumes that the left and right children of `p` have correct sums. Then we introduce a predicate `inv` (lines 25–26) to state that any node `p` verifies `good(p, 0)`, with the possible exception of node `except`. Using an exception is convenient to state that the invariant is locally violated during `upd_inv`. To state that the invariant holds for all nodes, we simply use `inv(NULL)`.

Our local invariant is convenient, as it does not require any induction. However, to convince the reader that we indeed proved the expected property, we also show that this local invariant implies a global, inductively-defined invariant. Lines 130–137 introduce the sum of all values in a tree, as an inductive predicate `treesum`, and a lemma to state that local invariant `inv(NULL)` implies `treesum`$(p, p{\rightarrow}\texttt{sum})$ for any node p.

4.3 Proof

The proof was performed using Frama-C Carbon[2] and its Jessie plug-in [21], using SMT solvers Alt-Ergo 0.92.3, CVC3 2.2, and Z3 2.19, on an Intel Core Duo 2.4 GHz. As explained in Sec. 3.4, we first run Frama-C on the annotated C code and then we insert the separation pragmas in the generated Jessie code (this is a benign modification). All verification conditions are discharged automatically within a total time of 30 seconds.

The two lemmas `parents_sep` and `global_invariant` were proved interactively using the Coq proof assistant version 8.3pl3 [26]. A total of 100 lines of tactics is needed. It doesn't take more than three days for one of the author to find the good specifications and make the proofs.

5 Function Footprints

In the case of the composite pattern, it is easy to specify the footprints of the C functions. Indeed, we can simply say that any `sum` field may be modified

[2] `http://frama-c.com/`

(using `\all->sum` in `assigns` clauses), since the invariant provides all necessary information regarding the contents of `sum` fields. For a function such as list reversal, however, we need to be more precise. We want to know that any list separated from the one being reversed is left unmodified. For instance, we would like to be able to prove the following piece of code:

```
1  /*@
2  requires islist(p) && islist(q) && sep_list_list(p,q);
3  ensures  islist(p) && islist(q) && sep_list_list(p,q);
4  @*/
5  void bar(struct node * p, struct node * q) {
6    p = rev(p);
7  }
```

For that purpose we must strengthen the specification and loop invariant of function `rev` with a suitable frame property. One possibility is to proceed as follows:

```
1  /*@
2  #Gen_Frame: list_frame list
3  #Gen_Sub: list_sub list list
4
5  requires list(p);
6  ensures  list(\result) && list_frame{Old,Here}(p,result);
7  @*/
8  struct node * rev(struct node * p);
9    ...
10   /*@ loop invariant
11         list(p) && list(q) && sep_list_list(p,q)
12         && list_frame{Init,Here}(\at(p,Init),q)
13         && list_sub{Init,Here}(\at(p,Init),p); @*/
14 ...
```

Two pragmas introduce new predicates `list_frame` and `list_sub`. Both depend on two memory states. The formula `list_frame{Old,Here}(p,result)` expresses in the post-condition that, between pre-state `Old` and post-state `Here`, all modified pointers belong to list `p`. It also specifies that the footprint of list `result` is included in the (old) footprint of list `p`. On the example of function `bar`, we now know that only pointers from `p` have been modified, so we can conclude that `islist(q)` is preserved. Additionally, we know that the footprint of `islist(p)` has not grown so we can conclude that it is still separated from `islist(q)`. The formula `list_sub{Init,Here}(\at(p,Init),p)` specifies only the inclusion of the footprint of the lists.

These two predicates could be axiomatized using membership only. For instance, `list_sub`(p, q) could be simply axiomatized as $\forall x, x \in \texttt{ft}_{\texttt{islist}}(p) \Rightarrow x \in \texttt{ft}_{\texttt{islist}}(q)$. But doing so has a rather negative impact on SMT solvers, as they have to first instantiate this axiom and then to resort to other axioms related to membership. Moreover this axiom is very generic and can be applied when not needed. For that reason we provide, in addition to axioms related to membership, axioms for footprint inclusion, to prove either $s \subset \texttt{ft}_p(p)$ or

$\texttt{ft}_p(p) \subset s$ directly. With such axioms, functions `rev` and `bar` are proved correct automatically.

6 Related and Future Work

VeriFast [13,15] allows user-defined predicates but requires user annotations to fold or unfold these predicates. In our work, we rely instead on the capability of first-order provers to fold and unfold definitions. VeriFast uses the SMT solver Z3, but only as a constraint solver on ground terms.

The technique of *implicit dynamic frames* [24] is closer to our work, except that formulas are restricted. Additionally, implicit dynamic frames make use of a set theory, whereas we do not require any, as we directly encode the relevant parts of set theory inside our footprint definition axioms.

Both these works do not allow a function to access (and thus modify) a pointer that is not in the footprint of the function's precondition — except if it is allocated inside the function. In our work, we do not have such a restriction. When necessary, we may define the footprint of a function using separation predicates, as explained in the first author's thesis [3].

There exist already several proofs of the composite pattern. One is performed using VeriFast [14]. It requires many lemmas and many `open`/`close` statements, whereas our proof does not contain much proof-related annotations.

The use of a local invariant in our proof is not new. It was first described in [19]. The proof by Rosenberg, Banerjee, and Naumann [23] also makes use of it. In order to prove that `addChild` is not creating cycles, the latter proof introduces two ghost fields, one for the set of descendants and one for the root node of the tree. Updating these ghost fields must be done at several places. In our case, we could manage to perform the case only with the generated predicate `sep_node_parents` without need of extra ghost fields which leads to a simpler proof.

The composite pattern has also been proved using *considerate reasoning* [25], a technique that advocates for local invariant like the one we used. Our predicate `inv` is similar to their `broken` declaration. As far as we understand, this proof is not mechanized, though.

Our future work includes generalizing the frame pragma used to describe the footprint of a function. One solution is to compute the footprint directly from ACSL's `assigns` clause, if any. Another is to describe the footprint using the linear maps framework [18]. One valuable future work would be to formally prove the consistency of our axioms, either using a meta-theoretical formalization, or, in a more tractable way, by producing proofs for each generated axiom.

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)

2. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
3. Bobot, F.: Logique de séparation et vérification déductive. Thèse de doctorat, Université Paris-Sud (December 2011)
4. Bobot, F., Conchon, S., Contejean, É., Iguernelala, M., Lescuyer, S., Mebsout, A.: The Alt-Ergo automated theorem prover (2008), http://alt-ergo.lri.fr/
5. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages, Wrocław, Poland (August 2011)
6. Bornat, R.: Proving Pointer Programs in Hoare Logic. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 102–126. Springer, Heidelberg (2000)
7. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, 23–50 (1972)
8. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. de Moura, L., Dutertre, B.: Yices: An SMT Solver, http://yices.csl.sri.com/
10. Filliâtre, J.-C., Paskevich, A., Stump, A.: The 2nd Verified Software Competition (November 2011), https://sites.google.com/site/vstte2012/compet
11. The Frama-C platform for static analysis of C programs (2008), http://www.frama-c.cea.fr/
12. Huisman, M., Klebanov, V., Monahan, R.: (October 2011), http://foveoos2011.cost-ic0701.org/verification-competition
13. Jacobs, B., Piessens, F.: The verifast program verifier. CW Reports CW520, Department of Computer Science, K.U.Leuven (August 2008)
14. Jacobs, B., Smans, J., Piessens, F.: Verifying the composite pattern using separation logic. In: Workshop on Specification and Verification of Component-Based Systems, Challenge Problem Track (November 2008)
15. Jacobs, B., Smans, J., Piessens, F.: A Quick Tour of the VeriFast Program Verifier. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 304–311. Springer, Heidelberg (2010)
16. Kassios, I.T.: Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
17. Klebanov, V., Müller, P., Shankar, N., Leavens, G.T., Wüstholz, V., Alkassar, E., Arthan, R., Bronish, D., Chapman, R., Cohen, E., Hillebrand, M., Jacobs, B., Leino, K.R.M., Monahan, R., Piessens, F., Polikarpova, N., Ridge, T., Smans, J., Tobies, S., Tuerk, T., Ulbrich, M., Weiß, B.: The 1st Verified Software Competition: Experience Report. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 154–168. Springer, Heidelberg (2011), Materials available at www.vscomp.org
18. Lahiri, S.K., Qadeer, S., Walker, D.: Linear maps. In: Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification, PLPV 2011, pp. 3–14. ACM, New York (2011)
19. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Aspects of Computing (2007)
20. Leino, K.R.M., Moskal, M.: VACID-0: Verification of ample correctness of invariants of data-structures, edition 0. In: Proceedings of Tools and Experiments Workshop at VSTTE (2010)

21. Moy, Y., Marché, C.: The Jessie plugin for Deduction Verification in Frama-C — Tutorial and Reference Manual. INRIA & LRI (2011), http://krakatoa.lri.fr/
22. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th Annual IEEE Symposium on Logic in Computer Science. IEEE Comp. Soc. Press (2002)
23. Rosenberg, S., Banerjee, A., Naumann, D.A.: Local Reasoning and Dynamic Framing for the Composite Pattern and Its Clients. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 183–198. Springer, Heidelberg (2010)
24. Smans, J., Jacobs, B., Piessens, F.: Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
25. Summers, A.J., Drossopoulou, S.: Considerate Reasoning and the Composite Design Pattern. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 328–344. Springer, Heidelberg (2010)
26. The Coq Development Team. The Coq Proof Assistant Reference Manual – Version V8.3 (2010), http://coq.inria.fr

A Annotated Source Code

```
1  typedef struct node {
2    int val, sum;
3    struct node *parent;
4    //@ ghost struct node *left, *right;
5  } *NODE;
6
7  /*@ inductive parents(NODE p) {
8    case nil: \forall NODE p; p==NULL ==> parents(p);
9    case cons: \forall NODE p;
10     p != NULL ==> \valid(p) ==>
11     parents(p->parent) ==> parents(p);
12 }
13
14 #Gen_Separation sep_node_parents(NODE, parents)
15
16 lemma parents_sep:
17   \forall NODE p; p!=NULL ==>
18     parents(p) ==> sep_node_parents(p, p->parent);
19
20 predicate good(NODE p, int delta) =
21   p->sum + delta == p->val +
22    (p->left == NULL? 0 : p->left->sum) +
23    (p->right == NULL? 0 : p->right->sum);
24
25 predicate inv(NODE except) =
26   \forall NODE p; \valid(p) ==> p != except ==> good(p, 0);
27
28 predicate wf(NODE except) =
29   \forall NODE p; \valid(p) ==> p != except ==>
30     (p->right != NULL ==>
31        p->right->parent == p && \valid(p->right)) &&
32     (p->left != NULL ==>
33        p->left->parent == p && \valid(p->left)) &&
34     (p->right == p->left ==> p->right == NULL) &&
```

```
35      (p->parent != NULL ==> \valid(p->parent)) &&
36      (p->parent != NULL ==>
37         p->parent->left == p || p->parent->right == p);
38
39  predicate newnode(NODE p, integer v) =
40         parents(p) && p->right == NULL && p->left == NULL &&
41         p->parent == NULL && p->val == v && \valid(p);
42  @*/
43
44  /*@ requires
45        inv(NULL) && wf(NULL);
46      ensures
47        inv(NULL) && wf(NULL) && newnode(\result, v) &&
48        \forall NODE n; \old(\valid(n)) ==>
49           \result != n && \valid(n) &&
50           \old(n->val) == n->val && \old(n->parent) == n->parent &&
51           \old(n->left) == n->left && \old(n->right) == n->right;
52  @*/
53  NODE create(int v) {
54   Before:
55      {
56    NODE p = (NODE)malloc(sizeof(struct node));
57    /*@ assert \forall NODE n; n != p ==>
58               \valid(n) ==> \at(\valid(n),Before); @*/
59    p->val = p->sum = v;
60    p->parent = p->left = p->right = NULL;
61    return p;
62  }}
63
64  /*@ requires inv(p) && parents(p) && wf(NULL) && good(p,delta);
65      ensures inv(NULL);
66      assigns \all->sum;
67   @*/
68  void upd_inv(NODE p, int delta) {
69     NODE n = p;
70     /*@ loop invariant
71           inv(n) && parents(n) && (n != NULL ==> good(n,delta));
72       @*/
73      while (n != NULL) {
74        n->sum = n->sum + delta;
75        n = n->parent;
76      }
77  };
78
79  /*@
80    requires
81      inv(NULL) && wf(NULL) &&
82      \valid(q) && q->parent == NULL &&
83      parents(p) && p != NULL && sep_node_parents(p, p->parent) &&
84      (p->left == NULL || p->right == NULL) && sep_node_parents(q,p);
85    ensures
86      parents(q) && parents(p) && inv(NULL) && wf(NULL) &&
87      (\old(p->left) == NULL ==>
88         p->left == q && \old(p->right) == p->right) &&
89      (\old(p->left) != NULL ==>
90         p->right == q && \old(p->left) == p->left);
91    assigns p->left, p->right, q->parent, \all->sum;
92  @*/
93  void addChild(NODE p, NODE q) {
94    if (p->left == NULL) p->left = q; else p->right = q;
```

```
95    q->parent = p;
96    upd_inv(p, q->sum);
97  }
98
99  /*@ requires parents(p) && p != NULL && inv(NULL) && wf(NULL);
100     ensures p->val == v && parents(p) && inv(NULL) && wf(NULL);
101     assigns p->val, \all->sum;
102 @*/
103 void update(NODE p, int v) {
104    int delta = v - p->val;
105    p->val = v;
106    upd_inv(p, delta);
107 }
108
109 /*@
110   requires
111     parents(p) && p != NULL && p->parent != NULL &&
112     inv(NULL) && wf(NULL);
113   ensures
114     parents(p) && p->parent == NULL && inv(NULL) && wf(NULL) &&
115     (\old(p->parent->left) == p ==>
116        \old(p->parent)->left == NULL) &&
117     (\old(p->parent->right) == p ==>
118        \old(p->parent)->right == NULL);
119   assigns p->parent->left, p->parent->right, p->parent, \all->sum;
120 @*/
121 void dislodge(NODE p) {
122   NODE n = p->parent;
123   if(p->parent->left == p) p->parent->left = NULL;
124   if(p->parent->right == p) p->parent->right = NULL;
125   p->parent = NULL;
126   upd_inv(n, -p->sum);
127 }
128
129 /*@
130 inductive treesum{L}(NODE p, integer v) {
131   case treesum_null{L}:
132     treesum(NULL, 0);
133   case treesum_node{L}:
134     \forall NODE p; p != NULL ==> \forall integer sl, sr;
135     treesum(p->left, sl) ==> treesum(p->right, sr) ==>
136     treesum(p, p->val + sl + sr);
137 }
138
139 lemma global_invariant{L}:
140   inv(NULL) ==> wf(NULL) ==>
141   \forall NODE p; \valid(p) ==> treesum(p, p->sum);
142 @*/
```

The Confinement Problem in the Presence of Faults*

William L. Harrison[1], Adam Procter[1], and Gerard Allwein[2]

[1] University of Missouri, MO 65211 USA
[2] US Naval Research Laboratory, Code 5543, Washington, DC, USA

Abstract. In this paper, we establish a semantic foundation for the safe execution of untrusted code. Our approach extends Moggi's computational λ-calculus in two dimensions with operations for asynchronous concurrency, shared state and software faults and with an effect type system *à* la Wadler providing fine-grained control of effects. An equational system for fault isolation is exhibited and its soundness demonstrated with a semantics based on monad transformers. Our formalization of the equational system in the Coq theorem prover is discussed. We argue that the approach may be generalized to capture other safety properties, including information flow security.

1 Introduction

Suppose that you possess an executable of unknown provenance and you wish to run it safely. The cost of analyzing the binary is prohibitive, and so, ultimately, you have little choice but to explore its effects by trial and error. That is, you run it and hope that nothing irreversibly damaging is done to your system. There are two alternatives proposed in the literature to the trial and error strategy. You can attempt to detect safety and security flaws in the untrusted code with automated static analyses. This is the approach being explored by much of the literature from the language-based security [27] community. The other approach is to isolate the untrusted code so that any destructive side effects (malicious or otherwise) resulting from its execution are rendered inert.

This paper introduces the *confinement calculus* (CC) and uses it as a vehicle for exploring the design and verification of isolation kernels (defined below). CC extends Moggi's computational λ-calculus [22] with constructs for state, faults and concurrency. Furthermore, the type system for the CC also incorporates an effect system *à* la Wadler [29] to distinguish computations occurring on different domains. The CC concurrency metalanguage is closely related to recent work of Goncharov and Schröder [12].

Lampson coined the term *confinement problem* [17] for the challenge of confining arbitrary programs—i.e., executing arbitrary code in a manner that prevents the illegitimate leakage of information through what Lampson termed *covert*

* This research was supported by NSF CAREER Award 00017806, US Naval Research Laboratory Contract 1302-08-015S, and by the U.S. Department of Education GAANN grant no. P200A100053.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 182–197, 2012.

channels. Legitimate channels transfer information via a system's resources used as intended by its architects. Covert channels transfer information by using or misusing system resources in ways unintended by the system's architects. The isolation property we define—called *domain isolation*—is similar to, albeit more restrictive than, the security property from our previous work [14]. Delimiting the scope of effects for arbitrary programs is the essence of confinement and the combination of effect types with monads is the scoping mechanism we use to confine effects.

A simple isolation kernel written in CC is presented in Figure 1. We assume there are two confinement domains, named Athens (A) and Sparta (S). The kernel is a function `k` which is parameterized by *domain handler functions* for each of the input domains, with types Δ_{A} and Δ_{S} respectively. These domain handlers are applied by the kernel to produce a single effectful computation step; the effect system guarantees that the effects of the Δ_{A} (resp. Δ_{S})-typed handler are restricted to A(S). The kernel also takes as input an internal kernel state value (here just a domain tag of type `D` which serves a similar function to a process id), and domain state values of types Dom_{A} and Dom_{S}. Execution of the respective threads is interleaved according to a round robin policy. The `unfold` operator encapsulates guarded recursion.

The proof that `k` is, in fact, an *isolation* kernel rests on two important features of the CC. The effect system guarantees that the domain handlers do not themselves induce state effects outside of their respective domains. The equational logic allows us to prove, using simple monadic equational reasoning, that the interleaving of the threads by `k` does not introduce new interactions between the domains: failure in Athens will not propagate to Sparta (nor vice versa).

```
D = {A, S}
k : (Δ_A × Δ_S) → (D × Dom_A × Dom_S) → R^D()
k (ha,hs) (s_0,α_0,σ_0) =
 unfold (s_0,α_0,σ_0)
        (λ(s,α,σ). case s of
              A -> case α of
                      (Just x) -> ha x >>= λα'. return (Left (S,α',σ))
                      Nothing  -> return (Left (S,α,σ))
              S -> case σ of
                      (Just x) -> hs x >>= λσ'. return (Left (A,α,σ'))
                      Nothing  -> return (Left (A,α,σ)))
```

Fig. 1. A Simple Isolation Kernel in CC

The structure of the remainder of this article is as follows. The rest of this section introduces the safety property *fault isolation* and motivates our approach to it. Section 2 presents an overview of the literature on effect systems and monads. The Confinement Calculus is defined in Section 3. Section 4 demonstrates how to use the CC to construct and verify an isolating kernel. Formalization of the Confinement Calculus in the Coq theorem prover is discussed in Section 5. Related work is discussed in Section 6, and Section 7 concludes.

A Monadic Analysis of Fault Isolation

Fault isolation is a safety property which prescribes boundaries on the extent of a fault effect. Here, we take a *fault* to mean a failure within a thread that causes it to terminate abnormally. The causes of a fault can be many and system-dependent, but some typical causes include activities such as division by zero, the corruption of a runtime stack, etc. Under some circumstances, one thread's failure can "crash" other threads. Fault isolation in this context means that the failure of one thread can only effect a subset of all threads running on a system.

We assume that the threads running on a system are partitioned into *domains* where the term is adapted from the terminology of hypervisors [3] and separation kernels [20] rather than denotational semantics. Fault isolation, as we use the term, means that a thread effect may only operate on its own domain. We refer to a *fault-isolating kernel* as a multitasking, multi-domain kernel in which the imperative and fault operations on one domain have no impact on other thread domains. Our fault model applies equally as well to software traps and programmable exceptions, although we do not provide the details here.

From the perspective of an individual thread, the scope of a fault should be global. Let the thread t be a sequence of atoms, $a_0; a_1; a_2; \cdots$, then, if a_0 causes a fault, then the execution of $a_1; a_2; \cdots$ should be cancelled, thereby satisfying the (pseudo-)equation, $a_0; a_1; a_2; \cdots = a_0$. From the point of view of a concurrent system (e.g., a multitasking kernel, etc.), the scope of a fault within an individual thread must remain isolated. The execution of t is really interwoven with other actions, including potentially those of other threads (e.g., $b_0; a_0; b_1; a_1; b_2; a_2; \cdots$), and a fault within t must not effect the execution of the other actions. In other words, should a_0 cause a fault, then the following (pseudo-)equation should hold, $b_0; a_0; b_1; a_1; b_2; a_2; \cdots = b_0; a_0; b_1; b_2; \cdots$, specifying that the subsequent actions of t should be filtered from the global system execution. The pseudo- prefix on the aforementioned equations signifies that the equations capture intuitions rather than rigorous mathematical truth. The confinement calculus will allow us to make these statements rigorous.

2 Effect Systems and Monads

Effect systems [24] and monads [22,18] are means of representing the potential side effects of a program explicitly within its type. This section provides a brief overview of effect systems and monads and motivates our use of their combination.

Effect Types. Effect systems are commonly associated with impure, strongly typed functional languages (e.g., ML [21]) because the effect type annotations make explicit the side effects already present implicitly in the language itself. In an impure, strongly-typed functional language, the type of a function specifies its input and output behavior only. An ML function, $f : \texttt{int} \to \texttt{int}$, takes and returns integer values, but, because ML is impure, it may also have side effects (e.g., destructive update or programmable exceptions) which are not reflected

in its type. An effect system would indicate the potential side effects in the type itself. Annotating the arrow in f's type with ρ (i.e., $f : \texttt{int} \xrightarrow{\rho} \texttt{int}$) could be used to indicate that f may destructively update region ρ. Effect annotations are introduced via side effecting language constructs (e.g., ML's assignment and dereference operations, `:=` and `!`, respectively). An effect type system tracks the effects within a program to indicate its potential side effects. For an excellent account of effect systems, the reader is referred to Nielson, et al. [24].

Monads. Pure, strongly-typed functional languages (e.g., Haskell [25]) do not allow side effects, so there are no implicit side effects to make explicit. Monads are used to mimic side effecting computations within a pure language. Monads in Haskell are type constructors with additional operations, bind (`>>=`) and unit (`return`), obeying the "monad laws" (defined in Figure 2). What makes monads useful is that programmers can tailor the desired effects to the application being constructed, effectively configuring a domain-specific language for each application. Rewriting it in Haskell, f now has type $\texttt{Int} \rightarrow \texttt{M Int}$ where `M` is the monad type constructor that encapsulates desired effects. Monads are also algebraic constructions with properties useful to formal verification (more will be said about this below). Figure 2 presents Moggi's well-known computational λ-calculus [22]. The computational λ-calculus is the core of any equational logic for monadic specifications, including the logic presented in Section 3. An equational judgment has the form, $\Sigma \vdash \Gamma \triangleright\ e_1 = e_2 : t$, where Σ, Γ and t are a set of hypotheses, a typing environment, and a type, respectively.

$$\frac{\Sigma \vdash \Gamma, x : B \triangleright\ e_1 = e_2 : A}{\Sigma \vdash \Gamma, x : B \triangleright\ \texttt{return}(e_1) = \texttt{return}(e_2) : MA} \quad \textsf{(cong1)}$$

$$\frac{\Sigma \vdash \Gamma, x : C \triangleright\ e_1 = e_2 : MA \quad \Sigma \vdash \Gamma, x' : A \triangleright\ e_1' = e_2' : MB}{\Sigma \vdash \Gamma, x : C \triangleright\ e_1 \texttt{>>=} \lambda x'.e_1' = e_2 \texttt{>>=} \lambda x'.e_2' : MB} \quad \textsf{(cong2)}$$

$$\frac{\Gamma, x : A \triangleright e_1 : MA \quad \Gamma, x_1 : B \triangleright e_2 : MB \quad \Gamma, x_2 : C \triangleright e_3 : MC}{\Sigma \vdash \Gamma, x : A \triangleright\ (e_1 \texttt{>>=} \lambda x_1.e_2) \texttt{>>=} \lambda x_2.e_3 = e_1 \texttt{>>=} \lambda x_1.(e_2 \texttt{>>=} \lambda x_2.e_3) : MC} \quad \textsf{(assoc)}$$

$$\frac{\Gamma, x : A \triangleright e_1 : B \quad \Gamma, x_1 : B \triangleright e_2 : MC}{\Sigma \vdash \Gamma, x : A \triangleright\ (\texttt{return}(e_1) \texttt{>>=} \lambda x_1.e_2) = [e_1/x_1]e_2 : MC} \quad \textsf{(l-unit)}$$

$$\frac{\Gamma, x : A \triangleright e_1 : MB}{\Sigma \vdash \Gamma, x : A \triangleright\ e_1 \texttt{>>=} \lambda x_1.\texttt{return}(x_1) = e_1 : MB} \quad \textsf{(r-unit)}$$

Fig. 2. The Computational λ-Calculus. M stands for any monad. The "monad laws" are assoc (associativity), l-unit (left unit), and r-unit (right unit).

Effect Systems + Monads. Combining effect systems with monadic semantics (as in Wadler [29]) provides fine-grained tracking of effects with a semantic model of those effects. Monads give rise to an integrated theory of effects and effect propagation. The integration of multiple effects within a single monad M has consequences for formal verification. Because all of the effects are typed in M, those effects are not distinguished syntactically within the type system of a

specification language. More positively, a rich equational theory governing their interaction follows by construction.

Effect systems can reflect this semantic information in the syntax of the specification language itself, thereby making monadic specifications more amenable to logical analysis. In the setting of this research, the combination of effects systems with monads is used to abstract over computations that occur on a particular domain. Given a particular domain d and monad K, for example, any term, $\Gamma \rhd e : K^{\{d\}} A$, is arbitrary code on domain d, i.e., its effects occur only in domain d. Combining effects systems and monads delimits the scope of effects for arbitrary programs and is the principal mechanism for designing and verifying confinement systems.

The Identity and State Monads. The identity (left) and state (right) monads are defined below (where *Sto* can be any type). The *return* operator is the monadic analogue of the identity function, injecting a value into the monad. The $\gg\!=$ operator is a form of sequential application. Monadic operators other than $\gg\!=$ and *return* are key to the formulation of a particular notion of computation. The state monad S encapsulates an imperative notion of computation with operators for updating and reading the state, u and g, resp.

$$
\begin{array}{ll}
\textbf{data}\ Id\ a & = Id\ a \\
return\ v & = Id\ v \\
(Id\ x) \gg\!= f & = f\ x \\
\textbf{data}\ Sa = S(Sto \to (a, Sto)) & \\
deS\ (S\ x) & = x \\
u : (Sto \to Sto) \to S() & \\
u\ f & = S\ (\lambda\sigma.((), f\ \sigma))
\end{array}
\qquad
\begin{array}{l}
g\ :\ S\ Sto \\
g\ =\ S\ (\lambda\sigma.(\sigma, \sigma)) \\
return\ v\ =\ S\ (\lambda\sigma.\ (v, \sigma)) \\
(S\ x)\ \gg\!= f \\
\quad =\ S\ (\lambda\sigma_0.\ \textbf{let}\ (v, \sigma_1)\ =\ x\ \sigma_0 \\
\qquad\qquad\qquad \textbf{in}\ deS\ (f\ v)\ \sigma_1)
\end{array}
$$

The Maybe Monad and Errors. The usual formulation of an error monad is called *Maybe* in Haskell (see below). An error (i.e., *Nothing*) has the effect of canceling the rest of the computation (i.e., `f`). The scope of *Nothing* is global in the sense that each of the following expressions evaluates to *Nothing*: $(Just\ 1 \gg\!= \lambda v.\ Nothing)$, $(Nothing \gg\!= \lambda d.\ Just\ 1)$, $(Just\ 1 \gg\!= \lambda v.\ Nothing \gg\!= \lambda d.\ Just\ 2)$.

$$
\begin{array}{ll}
\textbf{data}\ Maybe\ a = Just\ a \mid Nothing & \\
return & = Just
\end{array}
\qquad
\begin{array}{ll}
Just\ v \gg\!= f & = f\ v \\
Nothing \gg\!= f & = Nothing
\end{array}
$$

Observe that this behavior precludes the possibility of *fault isolation* within domains: if the *Nothing* occurs on one domain and the $(Just\ 1)$ or $(Just\ 2)$ occur on another, the entire multi-domain computation will be canceled. From a security point of view, this is clearly undesirable: allowing a high-security computation to terminate a low-security computation introduces information flow via a termination channel, and allowing a low-security computation to terminate a high-security computation exposes the system to a denial of service attack.

Monad Transformers. Monad transformers allow monads to be combined and extended. Monad transformers corresponding to the state monad are defined in Haskell below. Formulations of the state monad equivalent to those above are produced by the applications of this transformer to the identity monad, *StateT Sto Id*. In the following, type variable m abstracts over monads.

$$
\begin{array}{l}
\textbf{data}\ \mathit{StateT}\ s\ m\ a\ =\ \mathit{ST}\ (s\ \rightarrow\ m\ (a, s)) \\
\mathit{deST}\ (\mathit{ST}\ x)\ =\ x \\
\mathit{return}\ v\ =\ \mathit{ST}\ (\lambda s.\ \mathit{return}_m\ (v, s)) \\
(\mathit{ST}\ x)\ \gg\!=\ f\ =\ \mathit{ST}\ (\lambda s_0.\ (x\ s_0)\ \gg\!=_m\ \lambda(y, s1).\ \mathit{deST}\ (f\ y)\ s1) \\
\mathit{lift}\ :\ m\ a\ \rightarrow\ \mathit{StateT}\ s\ m\ a \\
\mathit{lift}\ \varphi\ =\ \mathit{ST}\ (\lambda s.\ \varphi\ \gg\!=_m\ \lambda v.\ \mathit{return}_m\ (v, s))
\end{array}
$$

For a monad m and type s, $(\mathit{StateT}\ s\ m)$ "extends" m with an updateable store s. The *lift* morphism is used to redefine any existing operations on m for the monad $(\mathit{StateT}\ s\ m)$. The process of lifting operations is analogous to inheritance in object-oriented languages. The layer $(\mathit{StateT}\ s\ m)$ also generalizes the definitions of the update and get operators:

$$
\begin{array}{ll}
u\ :\ (s\ \rightarrow\ s)\ \rightarrow\ \mathit{StateT}\ s\ m\ () & g\ :\ \mathit{StateT}\ s\ m\ s \\
u\ f\ =\ \mathit{ST}\ (\lambda s.\ \mathit{return}_m\ ((), f\ s)) & g\ =\ \mathit{ST}\ (\lambda s.\ \mathit{return}_m\ (s, s))
\end{array}
$$

Layered State Monads. A layered state monad is a monad constructed from multiple applications of the state monad transformer to an existing monad.

$$
\begin{array}{l}
\mathit{type}\ K\ =\ \mathit{StateT}\ \mathit{Sto}\ (\mathit{StateT}\ \mathit{Sto}\ (\mathit{StateT}\ \mathit{Sto}\ \mathit{Id})) \\
u_1,\ u_2,\ u_3\ :\ (\mathit{Sto}\ \rightarrow\ \mathit{Sto})\ \rightarrow\ K\ () \\
u_1\ f\ =\ u\ f \\
u_2\ f\ =\ \mathit{lift}\ (u\ f) \\
u_3\ f\ =\ \mathit{lift}\ (\mathit{lift}\ (u\ f))
\end{array}
$$

Each application of $(\mathit{StateT}\ \mathit{Sto})$ creates a layer with its own instances of the update (u_1-u_3) and get operations (not shown). These imperative operators come with useful properties by construction [14] and some of these are included as the equational rules (clobber) and (atomic n.i.) (atomic non-interference) in Section 3.

Resumption-Monadic Concurrency. Two varieties of resumption monad are utilized here, the *basic* and *reactive* resumption monads [13]. Basic resumptions encapsulate a concurrency-as-interleaving notion of computation, while reactive resumptions refine this notion to include a failure signal. The basic and reactive monad transformers are defined in Haskell in terms of monad m as:

$$
\begin{array}{l}
\textbf{data}\ \mathit{ResT}\ m\ a\ \ =\ \mathit{Done}\ a\ \mid\ \mathit{Pause}\ (m\ (\mathit{ResT}\ m\ a)) \\
\mathit{return}\ \ \ \ \ \ \ \ \ \ \ \ \ =\ \mathit{Done} \\
(\mathit{Done}\ v)\ \gg\!=\ f\ \ =\ f\ v \\
(\mathit{Pause}\ \varphi)\ \gg\!=\ f\ =\ \mathit{Pause}\ (\varphi\ \gg\!=_m\ \lambda\kappa.\ \mathit{return}_m\ (\kappa\ \gg\!=\ f))
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{ReactT}\ m\ a\ =\ \mathit{Dn}\ a\ \mid\ \mathit{Ps}\ (m\ (\mathit{ReactT}\ m\ a))\ \mid\ \mathit{Fail} \\
\mathit{return}\ \ \ \ \ \ \ \ \ \ \ \ \ \ =\ \mathit{Dn} \\
(\mathit{Dn}\ v)\ \gg\!=\ f\ \ \ \ \ \ =\ f\ v
\end{array}
$$

$$
\begin{array}{ll}
(Ps\ \varphi) \gg\!= f & = Ps\ (\varphi \gg\!=_m \lambda\kappa.\ return_m\ (\kappa \gg\!= f)) \\
Fail \gg\!= f & = Fail
\end{array}
$$

We chose to formulate the reactive resumption transformer along the lines of Swierstra and Altenkirch [28] rather than that of our previous work [13] because it is simpler. We define the following monads: $Re = ReactTK$, $R = ResTK$, and $K = StateT^n\ Sto\ Id$ where n is the the number of domains and Sto is the type of stores (left unspecified).

Figure 3 presents the concurrency and co-recursion operations for R and Re. The *step* operation lifts an m-computation into the R (resp. Re) monad, thereby creating an atomic (w.r.t. R (Re)) computation. A resumption computation may be viewed as a (possibly infinite) sequence of such steps; a finite R-computation will have the form, $(step_R\ m_1) \gg\!=_R \lambda v_1. \cdots \gg\!=_R \lambda v_n.\ (step_R\ m_n)$. The definition of $step_R$ is below ($step_{Re}$ is analogous). The $unfold_R$ operator is used to define kernels while the $unfold_{Re}$ operator is used to define threads. The co-recursion provided by these operators is the only form of co-recursion supported by the confinement calculus. An important consequence of this limitation on recursion is that it guarantees productivity [8]. It should be noted that the presence of *Maybe* in the type of $unfold_{Re}$ means that threads may fail, while its absence from the type of $unfold_R$ means that kernels cannot fail. The unfold operators are defined below; note that *Either a b* is simply Haskell-inspired notation for the sum type $a + b$.

```
step_R : K A → R A
step_R φ = Pause (φ >>= (return ∘ Done))

unfold_R : (Monad t) ⇒ a → (a → t (Either a b)) → ResT t b
unfold_R a f = step_R (f a) >>= λκ.
                 case κ of
                   (Left a')  → unfold_R a' f
                   (Right b)  → return b
unfold_Re : (Monad t) ⇒ a → (a → t (Maybe (Either a b))) → ReactT t b
unfold_Re a f = step_Re (f a) >>= λκ.
                  case κ of
                    (Just (Left a'))  → unfold_Re a' f
                    (Just (Right b))  → return b
                    Nothing           → Fail
```

Fig. 3. Monadic Concurrency and Co-recursion Operations

3 The Confinement Calculus

This section introduces the confinement calculus and defines its syntax, type system and semantics. The CC proceeds from Moggi's computational λ-calculus [22] and Wadler's marriage of type systems for effects with monads [29].

$$
\begin{array}{lll}
e, e' \in \mathit{Exp} & ::= & x \mid \lambda x.e \mid e\, e' \mid \texttt{return}\ e \mid e\ \texttt{>>=}\ \lambda x.e' \mid \texttt{get} \mid \texttt{upd}\ e \\
 & & \mid \texttt{fail} \mid \texttt{mask} \mid \texttt{out} \mid \texttt{step} \mid \texttt{unfold} \mid \texttt{zero} \mid \texttt{succ} \mid \texttt{natRec} \\
A, B \in \mathit{Type} & ::= & A \to B \mid \texttt{K}^{\sigma}\ A \mid \texttt{R}^{\sigma}\ A \mid \texttt{Re}^{\sigma}\ A \mid \texttt{Sto} \mid \texttt{Nat} \mid () \mid A + B \mid A \times B
\end{array}
$$

Fig. 4. *Abstract Syntax.* Assume $D = \{d_1, \ldots, d_n\}$ and $\sigma \in \mathcal{P}(D)$.

Types in the CC are directly reflective of semantic domains introduced in the previous section, and as a result are named similarly. As a notational convention, we will use teletype font when expressing a type in CC (e.g. `K Nat`), and an italic font when referring to semantic domains (e.g. $K\ Nat$).

Abstract Syntax. Figure 4 presents the abstract syntax for the CC. The finite set of domains, D contains labels for all thread domains in the system (D replaces *Region* from Wadler's original presentation of MONAD [29]). We also diverge slightly from Wadler's language in that we do not track the "sort" of effects that a computation may cause: reading, writing, and failure are all treated the same.

The monadic expression language *Exp* has familiar computational λ-calculus constructs as well as imperative operations (`get, upd`), an imperative operation (`mask`) used in specifying the isolation of imperative effects [14], and others for resumption monadic computations (`out`, `step`, and `unfold`). Intuitively, the `mask` operation has the effect of resetting or "zeroing out" a particular domain. Expressions `unfold`, `step` and `out` are resumption-monadic operations. The `unfold` operator encapsulates corecursion, and its semantics are structured to allow only guarded recursion. More will be said about `fail`, `step` and `out` later in this section. Finally, the expressions `zero`, `succ`, and `natRec` allow construction of, and primitive recursion over, natural numbers. Note that the *only* forms of recursion permitted by CC are primitive recursion over naturals (via `natRec`), and guarded corecursion over resumptions (via `unfold`).

$$
\frac{\Gamma \triangleright e : A}{\Gamma \triangleright \texttt{return}\ e : \texttt{K}^{\emptyset} A}
\qquad
\frac{\Gamma \triangleright e : \texttt{K}^{\sigma} A \quad \Gamma, x : A \triangleright e' : \texttt{K}^{\sigma'} B}{\Gamma \triangleright e\ \texttt{>>=}\ \lambda x.e' : \texttt{K}^{\sigma \cup \sigma'} B}
\qquad
\frac{\Gamma \vdash e : \texttt{K}^{\sigma} A \quad \sigma \subseteq \sigma'}{\Gamma \vdash e : \texttt{K}^{\sigma'} A}
$$

$$
\frac{}{\Gamma \vdash \texttt{get} : \texttt{K}^{\{d\}} \texttt{Sto}}
\qquad
\frac{\Gamma \vdash f : \texttt{Sto} \to \texttt{Sto}}{\Gamma \vdash \texttt{upd}\ f : \texttt{K}^{\{d\}} ()}
\qquad
\frac{}{\Gamma \triangleright \texttt{mask} : \texttt{K}^{\{d\}} ()}
$$

Fig. 5. Type System for Imperative Effects

The type syntax in Figure 4 contains three monads. The monads `K`, `R` and `Re` encapsulate layered state, concurrency and system executions, and concurrent threads, respectively. The effect system can express fine-grained distinctions about computations and, in particular, allows the domain of a thread to be expressed in its type.

Types and Effects for the Confinement Calculus. The type and effect system for the CC is presented in Figures 5-7 and that figure is divided into

$$\frac{\Gamma \triangleright e : A}{\Gamma \triangleright \texttt{return}\ e : \texttt{R}^{\emptyset} A} \qquad \frac{\Gamma \triangleright e : \texttt{R}^{\sigma} A \quad \Gamma, x : A \triangleright e' : \texttt{R}^{\sigma'} B}{\Gamma \triangleright e\ \texttt{>>=}\ \lambda x.e' : \texttt{R}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \triangleright e : \texttt{K}^{\sigma} A}{\Gamma \triangleright \texttt{step}\ e : \texttt{R}^{\sigma} A}$$

$$\frac{\Gamma \triangleright p : \texttt{R}^{\sigma} A}{\Gamma \triangleright \texttt{out}\ p : \texttt{K}^{\sigma}(\texttt{R}^{\sigma} A)} \qquad \frac{\Gamma \triangleright p : A \quad \Gamma, x{:}A \triangleright q : \texttt{K}^{\sigma}(A + B)}{\Gamma \triangleright \texttt{unfold}\ p\ (\lambda x.q) : \texttt{R}^{\sigma} B} \qquad \frac{}{\Gamma \triangleright \texttt{natRec} : A{\to}(A{\to}A){\to}Nat{\to}A}$$

Fig. 6. Type System for Concurrency. The rule for `natRec` is also included.

three sections. Figure 5 contains the system for the imperative core of CC—i.e., the computations in the monad `K`. Figure 6 contains the type system for concurrency. Figure 7 contains the type system for threads—i.e., computations in the `Re` monad. The `Re` monad expresses the same notion of computation as the `R` monad, except that it also contains a signal `fail`. A thread may use `fail` to generate a fault, and it is up to the kernel component to limit the extent of the fault to the thread's domain.

$$\frac{\Gamma \triangleright e : A}{\Gamma \triangleright \texttt{return}\ e : \texttt{Re}^{\emptyset} A} \qquad \frac{\Gamma \triangleright e : \texttt{Re}^{\sigma} A \quad \Gamma, x : A \triangleright e' : \texttt{Re}^{\sigma'} B}{\Gamma \triangleright e\ \texttt{>>=}\ \lambda x.e' : \texttt{Re}^{\sigma \cup \sigma'} B} \qquad \frac{\Gamma \triangleright e : \texttt{K}^{\sigma} A}{\Gamma \triangleright \texttt{step}\ e : \texttt{Re}^{\sigma} A}$$

$$\frac{\Gamma \triangleright p : \texttt{Re}^{\sigma} A}{\Gamma \triangleright \texttt{out}\ p : \texttt{K}^{\sigma}(\texttt{Re}^{\sigma} A)} \qquad \frac{\Gamma \triangleright p : A \quad \Gamma, x{:}A \triangleright q : \texttt{K}^{\sigma}(A + B + ())}{\Gamma \triangleright \texttt{unfold}\ p\ (\lambda x.q) : \texttt{Re}^{\sigma} B} \qquad \frac{}{\Gamma \triangleright \texttt{fail} : \texttt{Re}^{\{d\}} A}$$

$$\frac{A <: B \qquad \sigma_0 \subseteq \sigma_1}{\texttt{K}^{\sigma_0} A <: \texttt{K}^{\sigma_1} B} \qquad \frac{A <: B \qquad \sigma_0 \subseteq \sigma_1}{\texttt{R}^{\sigma_0} A <: \texttt{R}^{\sigma_1} B} \qquad \frac{A <: B \qquad \sigma_0 \subseteq \sigma_1}{\texttt{Re}^{\sigma_0} A <: \texttt{Re}^{\sigma_1} B}$$

Fig. 7. Type System for Reactive Concurrency; Subtyping Relation

Figure 7 gives the rules for subtyping monadic computations (standard rules for reflexivity, transitivity, and for arrow, product, and sum types are omitted). The intuition is that one monadic computation φ may stand in for another computation γ without breaking type safety, if and only if the result type of φ is a subtype of that of γ, *and* φ's affected domains are a subset of γ's. An effect-free computation of type $\texttt{K}^{\emptyset} A$ also has type $\texttt{K}^{\{d\}} A$ (but not vice versa).

Denotational Semantics of the Confinement Calculus. The dynamic semantics of CC is a typed denotational semantics, meaning that the denotation of terms depends in part on their typing derivations. This allows us to overload the monad operations. The denotation of types does not depend on effect annotations and is completely standard. The `out` operation accesses the first step of a concurrent (`R`-typed) computation, producing a `K`-typed computation. Omitted from Figure 8 is the denotation of `natRec`, which is defined by structural induction on naturals. The CC term `run` and its denotation are defined as:

$$\texttt{run n } \varphi_0 \texttt{ = natRec (return}_K\ \varphi_0\texttt{) } (\lambda\varphi.\ (\varphi \texttt{ >>= out}_R\texttt{)) n}$$

$$\begin{aligned}
&run : Nat \to RA \to K(RA)\\
&run\ 0\ \varphi \quad\ = return\,\varphi\\
&run\ (n+1)\ \varphi = out\ \varphi\ \gg\!= run\ n
\end{aligned}$$

$$\begin{aligned}
&[\![\Gamma \triangleright x : A]\!]\rho && = \rho\ x\\
&[\![\Gamma \triangleright \lambda x.e : A \to B]\!]\rho && = \lambda v.[\![\Gamma, x : A \triangleright e : B]\!](\rho[x \mapsto v])\\
&[\![\Gamma \triangleright e\ e' : A]\!]\rho && = ([\![\Gamma \triangleright e : B \to A]\!]\rho)\ ([\![\Gamma \triangleright e' : B]\!]\rho)\\
&[\![\Gamma \triangleright \texttt{return}\ e : M^{\emptyset} A]\!]\rho && = return_M\ ([\![\Gamma \triangleright e : A]\!]\rho)\\
&[\![\Gamma \triangleright \texttt{get} : \texttt{K}^{d_i}\ Sto]\!]\rho && = lift_i\ g\\
&[\![\Gamma \triangleright \texttt{upd}\ \delta : \texttt{K}^{d_i}\ ()]\!]\rho && = lift_i\ (u\ ([\![\Gamma \triangleright \delta : Sto \to Sto]\!]\rho))\\
&[\![\Gamma \triangleright \texttt{mask} : \texttt{K}^{d_i}\ ()]\!]\rho && = lift_i\ (u\ (\lambda x.s_0))\\
&[\![\Gamma \triangleright e \texttt{ >>= } \lambda x.e' : M^{\sigma \cup \sigma'} B]\!]\rho\\
&\qquad = [\![\Gamma \triangleright e : M^{\sigma} A]\!]\rho \texttt{>>=}_M \lambda v.[\![\Gamma, x{:}A \triangleright e' : M^{\sigma'} B]\!](\rho[x \mapsto v])\\
&[\![\Gamma \triangleright \texttt{unfold}\ e\ (\lambda x.e') : \texttt{R}^{\sigma} B]\!]\rho\\
&\qquad = unfold_{\texttt{R}}\ ([\![\Gamma \triangleright e : A]\!]\rho)\ (\lambda v.[\![\Gamma, x : A \triangleright e' : \texttt{K}^{\sigma}(A + B)]\!]\ (\rho[x \mapsto v]))\\
&[\![\Gamma \triangleright \texttt{out}(p) : \texttt{K}^{\sigma}(\texttt{R}^{\sigma} A)]\!]\rho && = \begin{cases} return\,(Done\ v) & \text{if } [\![p]\!]\rho = Done\ v\\ \varphi & \text{if } [\![p]\!]\rho = Pause\ \varphi \end{cases}\\
&[\![\Gamma \triangleright \texttt{fail} : \texttt{Re}^{d_i}\ A]\!]\rho && = Fail\\
&[\![\Gamma \triangleright \texttt{unfold}\ e\ (\lambda x.e') : \texttt{Re}^{\sigma} B]\!]\rho\\
&\qquad = unfold_{Re}\ ([\![\Gamma \triangleright e : A]\!]\rho)(\lambda v.[\![\Gamma, x : A \triangleright e' : \texttt{K}^{\sigma}(A + B + ())]\!]\ (\rho[x \mapsto v]))\\
&[\![\Gamma \triangleright \texttt{out}(p) : \texttt{K}^{\sigma}(\texttt{Re}^{\sigma} A)]\!]\rho = \begin{cases} return\,(Just(Dn\ v)) & \text{if } [\![p]\!]\rho = Dn\ v\\ \varphi \gg\!= (return \circ Just) & \text{if } [\![p]\!]\rho = Ps\ \varphi\\ return\,Nothing & \text{if } [\![p]\!]\rho = Fail \end{cases}
\end{aligned}$$

Fig. 8. Denotational Semantics. M stands for the K, R, or Re monads. K, R, and Re are defined in Section 2.

Equational Logic. The rules of the equational logic encode known facts about the denotational semantics proven in an earlier publication [14]. For instance, the *run* operator "unrolls" R computations:

$$run\ (n+1)\ (step\ \varphi \gg\!=_R f) = \varphi \gg\!=_K run\ n \circ f \tag{1}$$

$$run\ (n+1)\ (return_R\ x) = return_K\ (return_R\ x) \tag{2}$$

Properties (1) and (2) justify our introduction of the following rules:

$$\frac{\Gamma \triangleright n : Nat \quad \Gamma \triangleright \varphi : \texttt{K}^{\sigma} A \quad \Gamma, x : A \triangleright e : \texttt{R}^{\sigma} B}{\Sigma \vdash \Gamma \triangleright\ \texttt{run}\ (n{+}1)\ (\texttt{step}\ \varphi\ \texttt{>>=}\ \lambda x.e) = \varphi\ \texttt{>>=}\ \lambda x.\texttt{run}\ n\ e\ :\ \texttt{K}^{\sigma}(\texttt{R}^{\sigma} B)} \quad \text{(run-step)}$$

$$\frac{\Gamma \triangleright n : Nat \quad \Gamma \triangleright e : A}{\Sigma \vdash \Gamma \triangleright\ \texttt{run}\ (n{+}1)\ (\texttt{return}_R\ e) = \texttt{return}_K\,(\texttt{return}_R\ e)\ :\ \texttt{K}^{\sigma}(\texttt{R}^{\sigma}\ A)} \quad \text{(run-return)}$$

A straightforward induction on the structure of type derivations justifies the soundness of the following rules. This induction makes use of previous work

(specifically Theorems 1-3 on page 17 [14]) and the "lifting law" of Liang [18]: $lift(x \gg\!= f) = liftx \gg\!= lift \circ f$.

$$\frac{\Gamma \triangleright \varphi : \mathtt{K}^{\sigma_0} A \qquad \Gamma \triangleright \mathtt{mask} : \mathtt{K}^{\sigma_1}() \qquad \sigma_0 \subseteq \sigma_1}{\Sigma \vdash \Gamma \triangleright \ \varphi \mathbin{>>} \mathtt{mask} = \mathtt{mask} \ : \ \mathtt{K}^{\sigma_1}()} \quad \text{(clobber)}$$

$$\frac{\Gamma \triangleright \varphi : \mathtt{K}^{\sigma_0}() \qquad \Gamma \triangleright \gamma : \mathtt{K}^{\sigma_1}() \qquad \sigma_0 \cap \sigma_1 = \emptyset}{\Sigma \vdash \Gamma \triangleright \ \varphi \mathbin{>>} \gamma = \gamma \mathbin{>>} \varphi \ : \ \mathtt{K}^{\sigma_0 \cup \sigma_1}()} \quad \text{(atomic n.i.)}$$

4 Isolation Kernels in Confinement Calculus

In this section, we turn our attention the construction of *isolation kernels* within the confinement calculus. An isolation kernel is a function which interleaves the execution of two or more threads in different domains, without introducing any interactions across domains. Put another way, an isolation kernel must have the property that a computation in domain d, when interleaved with a computation in $d' \neq d$, behaves exactly the same as it would if the d' computation had never happened.

The formal definition of isolation is made in terms of a notion called *domain similarity*. Two computations φ and γ are domain similar in a domain d if and only if for every finite prefix of φ, there exists a finite prefix of γ whose effects in d are the same.

Definition 1 (Domain Similarity Relation). *Consider two computations* $\varphi, \gamma : \mathtt{R}^{d_1 \cup \ldots \cup d_n} A$. *We say* φ *and* γ *are* similar with respect to domain d_i (*written* $\varphi \sim_{d_i} \gamma$) *if and only if the following holds.*

$$\forall\, n{\in}N.\, \exists\, m{\in}N.\, \mathtt{run}\, n\, \varphi \mathbin{>>_K} \mathtt{mask} \;=\; \mathtt{run}\, m\, \gamma \mathbin{>>_K} \mathtt{mask}$$

where $\mathtt{mask} = \mathtt{mask}_{d_1} \mathtt{>>} \ldots \mathtt{>>} \mathtt{mask}_{d_{i-1}} \mathtt{>>} \mathtt{mask}_{d_{i+1}} \mathtt{>>} \ldots \mathtt{>>} \mathtt{mask}_{d_n}$.

Kernels. Assume in Definitions 2-4 that $D = \{d_1, \ldots, d_n\}$ is a fixed set of domains. We first define a notion of *state*: namely a tuple containing one element representing the kernel's internal state, and an additional element for the state of each confinement domain. The domain states are wrapped in a *Maybe* constructor to represent the possibility of failure within a domain.

Definition 2 (Domain and Kernel State). *The state of domain* i, Dom_i *is defined as:*

$$Dom_i = \mathtt{Re}^{\{d_i\}}()$$

The type of kernel states for a type t *is:*

$$S = t \times (\mathtt{Maybe}\ Dom_1) \times \ldots \times (\mathtt{Maybe}\ Dom_n)$$

A kernel is parameterized by *handlers* for each domain. A handler is a state transition function on domain states, which may also have effects on the global state (hence the presence of K in the type). The only restriction on a handler for domain d_i is that its effects must be restricted to d_i.

Definition 3 (Domain Handler Function). *The type of handler functions for domain d_i is:*

$$\Delta_{d_i} = Dom_i \to \mathtt{K}^{\{d_i\}}(\mathtt{Maybe}\ Dom_i)$$

The handler vector type for D is:

$$\Delta_D = \Delta_1 \times \ldots \times \Delta_n$$

Putting the pieces together brings us to the definition of a kernel. Note that a kernel in our sense is parameterized over handler vectors.

Definition 4 (Kernel). *Given a handler vector Δ_D, kernel state type S, and an answer type Ans, the type of kernels is defined by the following:*

$$\Delta_D \to S \to \mathtt{K}^D(S + Ans)$$

Defining and Proving Isolation. To simplify the presentation, we will restrict our attention for the remainder of this section to the case of two domains, called A (for *Athens*) and S (for *Sparta*). All the results here generalize naturally to more than two domains.

We define isolation by an extensional property on kernels. A kernel is said to be isolating if the result of "eliminating" the computation in any one domain has no effect on the outcome of any other. This is a property akin to noninterference [11]. We express this formally by replacing the domain handler with `return` – the "do-nothing" computation – and replacing the domain state with `Nothing`.

Definition 5 (Isolation in the Presence of Faults). *A kernel k is* isolating in the presence of faults *if and only if*

$$k\ (f_\mathsf{A}, f_\mathsf{S})(s, d_\mathsf{A}, d_\mathsf{S}) \sim_\mathsf{A} k\ (f_\mathsf{A}, \mathtt{return})(s, d_\mathsf{A}, \mathtt{Nothing})$$
$$k\ (f_\mathsf{A}, f_\mathsf{S})(s, d_\mathsf{A}, d_\mathsf{S}) \sim_\mathsf{S} k\ (\mathtt{return}, f_\mathsf{S})(s, \mathtt{Nothing}, d_\mathsf{S})$$

The following theorem shows that kernel `k` of Figure 1 satisfies this definition.

Theorem 1 (`k` is isolating). *The kernel `k` of Figure 1 is isolating in the presence of faults.*

Proof. We will show the S-similarity side of the proof, since the A-similarity proof is analogous. N.b., Definition 1 allows the number of steps on each side of the equation (n and m) to be different. Here it suffices to fix $m = n$:

$$\mathtt{run}\ n\ (\mathtt{k}\ (f_\mathsf{A}, f_\mathsf{S})\ (s, d_\mathsf{A}, d_\mathsf{S}))\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A}$$
$$= \mathtt{run}\ n\ (\mathtt{k}\ (\mathtt{return}, f_\mathsf{S})\ (s, \mathtt{Nothing}, d_\mathsf{S}))\ \mathtt{>>}\ \mathtt{mask}_\mathsf{A}$$

The proof is by induction on n. The base case is trivial. We proceed by cases on s, d_A, and d_S. The most interesting case is when $s = \mathsf{A}$ and $d_\mathsf{A} = \mathtt{Just}\ x$; all others involve no more than straightforward evaluation and an application of the induction hypothesis. Let $s = \mathsf{A}$ and $d_\mathsf{A} = \mathtt{Just}\ x$. Then:

$$\mathtt{run}(n+1)(\mathtt{k}(f_\mathsf{A}, f_\mathsf{S})(\mathsf{A}, \mathtt{Just}\ x, d_\mathsf{S}))\mathtt{>>}\mathtt{mask}_\mathsf{A}$$

$$
\begin{array}{ll}
= \mathtt{run}\,1\ (\mathtt{k}\ (f_{\mathsf{A}}, f_{\mathsf{S}})\ (\mathsf{A}, \mathtt{Just}\ \ x, d_{\mathsf{S}}))\ \mathtt{>>=}\ \mathtt{run}\,n\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{prop run}\} \\
= f_{\mathsf{A}}\ x\ \mathtt{>>=}\ \lambda d'_{\mathsf{A}}.\ \mathtt{run}\,n\ (\mathtt{k}\ (f_{\mathsf{A}}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Just}\ \ x, d_{\mathsf{S}}))\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{evaluation}\} \\
= f_{\mathsf{A}}\ x\ \mathtt{>>=}\ \lambda d'_{\mathsf{A}}.\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}}))\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{i.h.}\} \\
= f_{\mathsf{A}}\ x\ \mathtt{>>=}\ \lambda d'_{\mathsf{A}}.\ \mathtt{mask}_{\mathsf{A}}\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) & \{\text{atomic n.i.}\} \\
= f_{\mathsf{A}}\ x\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}}\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) & \{d'_{\mathsf{A}} \text{ does not occur}\} \\
= \mathtt{mask}_{\mathsf{A}}\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}})) & \{\text{clobber}\} \\
= \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}}))\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{atomic n.i.}\} \\
= \mathtt{return}\ ()\ \mathtt{>>}\ \mathtt{run}\,n\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{S}, \mathtt{Nothing}, d_{\mathsf{S}}))\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{left unit}\} \\
= \mathtt{run}\,1\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{A}, \mathtt{Nothing}, d_{\mathsf{S}}))\ \mathtt{>>=}\ \mathtt{run}\,n\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{evaluation}\} \\
= \mathtt{run}\,(n+1)\ (\mathtt{k}\ (\mathtt{return}, f_{\mathsf{S}})\ (\mathsf{A}, \mathtt{Nothing}, d_{\mathsf{S}}))\ \mathtt{>>}\ \mathtt{mask}_{\mathsf{A}} & \{\text{prop run}\}
\end{array}
$$

5 Mechanizing the Logic in Coq

The syntax, denotational semantics, and equational logic of CC have all been mechanized in the Coq [8] theorem prover. In lieu of separate syntaxes for type judgments and terms as presented in the preceding sections, the Coq formulation uses a strongly-typed term formulation as suggested by Benton et al [7]. We combine this with a dependently typed denotational semantics along the lines of Chlipala [9] (see Chapter 9). The payoff of this approach is that we do not need to establish many of the usual properties such as progress and subject reduction that usually accompany an operational approach. Furthermore, strongly-typed terms are much more amenable to the use of Coq's built-in system of parametric relations and morphisms. Just a handful of relation and morphism declarations lets us reuse many of Coq's standard tactics (e.g., `replace` and `rewrite`) when reasoning in the CC logic.

Figure 9 presents an example equational judgment rule from the Coq development, representing the clobber rule. The only major difference between the Coq formalism and the corresponding rule in Section 3 has to do with the need for explicit subtyping: the term constructor `subsume` casts a term from a supertype to a subtype, and requires as an argument a proof term showing that the subtyping relationship holds (here called `S_just`). The full development in Coq is available by request.

6 Related Work

Klein et al. [15] describe their experience in designing, implementing and verifying the seL4 secure kernel. Monads are applied as an organizing principle at all levels of the seL4 design, implementation and verification. Their model of effects

```
J_clobber : ∀ (Γ:list ty) (d:domain) (t:ty)
              (te:term Γ (tyK (onedom d) t)),
   let S_just := S_tyK (onedom d) (union (onedom d) (onedom d))
                       (S_refl tynil) (clobber_obligation _)
   in  eq_judgment (subsume S_just (nullbindK te (mask Γ d))) (mask Γ d)
```

Fig. 9. Expressing the clobber rule in Coq

is different from the one described here. The seL4 monadic models encapsulate errors, state and non-determinism. The notions of computation applied here include concurrency and interactivity (R and Re, respectively) as well as layered state (K). Also, the type system underlying the seL4 models does not include effect types. The present work models faults via simulation on distinct domains rather than as part of an integrated model of effects. Cock, Klein and Sewell [10] apply Hoare-style reasoning to prove that a design is fault free. Kernels in the CC are fault-free as a by-product of our type system and it would be interesting to investigate whether the application of CC in the seL4 construction and verification process would alleviate some verification effort.

Another similarly interesting question is to consider the impact of integrating layered state and resumption-based concurrency into their abstract, executable and machine models. One design choice, for example, concerns the placement of preemption points—i.e., places where interrupts may occur—within the seL4 kernel specifications. It seems plausible that interrupt handling in seL4 might be simplified by the presence of an explicit concurrency model—i.e., resumptions. It also seems plausible that aspects of the seL4 design and verification effort might be reduced or abstracted by the inclusion of effect-scoping mechanisms like layered state and effect types—e.g., issues arising from the separation of kernel space from user space.

Language-based security [27] seeks to apply concepts from programming languages research to the design, construction and verification of secure systems. While fault isolation is generally considered a safety property, it is also a security property as well in that an unconfined fault may be used for a denial of service attack and also as a covert channel. Monads were first applied within the context of language-based security by Abadi et al. [1], although the use of effect systems seems considerably less common in the security literature. Bartoletti et al. [4,5] apply effect systems to history-based access control and to the secure orchestration of networked services. Bauer et al. [6] applied the combination of effect systems and monads to the design of secure program monitors; their work appears to be the first and only previously published research in security to do so. The current work differs from theirs mainly in our use of interaction properties of effects that follow by the construction of the monads themselves. These by-construction properties provide considerable leverage towards formal verification. Scheduler-independent security [26,19]) considers the relationship between scheduling and security and investigates possibilistic models of security that do not depend on particular schedulers. A natural next step for the current research is to investigate scheduler freedom with respect to CC kernels (e.g., Definition 4).

7 Conclusions

The research described here seeks to apply tools and techniques from programming languages research—e.g., monads, type theory, language compilers—to the production of high assurance systems. We are interested, in particular, in reducing the cost of certification and re-certification of verified artifacts. The questions we

confront are, in terms of semantic effects, what can untrusted code do and, given a semantic model of untrusted code, how can we specify a system for running it safely in isolation? Untrusted code can read and write store obviously. It can fail—i.e., cause an exception. It can also signal the operating system via a trap. These effects are inherited from the machine language of the underlying hardware.

Previous work [14] explored the application of modular monadic semantics to the design and verification of separation kernels. Each domain is associated with an individual state monad transformer [18] and the "layered" state monad constructed with these transformers has "by-construction" properties that are helpful for verifying the information flow security. The present work builds upon this in the following ways. Our previous work did not consider fault effects, and the layering approach taken in that work does not generalize to handle faults. It is also interesting, albeit less significant, that the semantics of the CC effect system is organized by state monad transformers. Structuring the semantics of Wadler's MONAD language with monad transformers was first suggested by Wadler [29] and the present work, to the best of our knowledge, is the first to actually do so. Although the current work focuses on isolation, we believe that it can be readily adapted to MILS (multiple independent levels of security) systems by refining the effect types to distinguish, for example, reads and writes on domains (as Wadler's original MONAD language did). One could then express, for example, a high security handler that is allowed to read from, but not write to, domains lower in the security hierarchy.

Kobayashi [16] proposed a general framework for reasoning about monadic specifications based in modal logic in which monads are formalized as individual modalities. Nanevski elaborated on the monad-as-modality paradigm, introducing a modal logic for exception handling as an alternative to the exception monad [23]. Nanevski's logic is an S4 modal logic in which necessity encodes a computation which may cause an exception—i.e., $\Box_C A$ represents a computation of an A value that may cause an exception named in set C. Modal logics have been adapted to security verification by partially ordering modalities to reflect a security lattice by Allwein and Harrison [2]. We are currently investigating the integration of the monad-as-modality paradigm with security-enabled partially-ordered modalities into a modal logic for verifying the security of monadic specifications in the confinement calculus and related systems.

References

1. Abadi, M., Banerjee, A., Heintze, N., Riecke, J.: A core calculus of dependency. In: 26th ACM Symp. on Principles of Programming Languages, pp. 147–160 (1999)
2. Allwein, G., Harrison, W.L.: Partially-ordered modalities. In: Advances in Modal Logic, pp. 1–21 (2010)
3. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 164–177 (2003)
4. Bartolett, M., Degano, P., Ferrari, G.-L.: History-Based Access Control with Local Policies. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 316–332. Springer, Heidelberg (2005)

5. Bartoletti, M., Degano, P., Ferrari, G.L.: Types and effects for secure service orchestration. In: 19th IEEE Computer Security Foundations Workshop (2006)
6. Bauer, L., Ligatti, J., Walker, D.W.: Types and Effects for Non-interfering Program Monitors. In: Okada, M., Babu, C.S., Scedrov, A., Tokuda, H. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 154–171. Springer, Heidelberg (2003)
7. Benton, N., Hur, C.-K., Kennedy, A., McBride, C.: Strongly Typed Term Representations in Coq. Journal of Automated Reasoning, 1–19 (to appear)
8. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
9. Chlipala, A.: Certified programming with dependent types Book draft of April 12 (2012), http://adam.chlipala.net/cpdt/
10. Cock, D., Klein, G., Sewell, T.: Secure Microkernels, State Monads and Scalable Refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 167–182. Springer, Heidelberg (2008)
11. Goguen, J., Meseguer, J.: Security policies and security models. In: Symposium on Security and Privacy, pp. 11–20. IEEE (1982)
12. Goncharov, S., Schröder, L.: A Coinductive Calculus for Asynchronous Side-Effecting Processes. In: Owe, O., Steffen, M., Telle, J.A. (eds.) FCT 2011. LNCS, vol. 6914, pp. 276–287. Springer, Heidelberg (2011)
13. Harrison, W.L.: The Essence of Multitasking. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 158–172. Springer, Heidelberg (2006)
14. Harrison, W.L., Hook, J.: Achieving information flow security through monadic control of effects. Journal of Computer Security 17(5), 599–653 (2009)
15. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP), pp. 207–220 (2009)
16. Kobayashi, S.: Monad as modality. Theor. Computer Science 175(1), 29–74 (1997)
17. Lampson, B.: A note on the confinement problem. CACM 16(10), 613–615 (1973)
18. Liang, S.: Modular Monadic Semantics and Comp. PhD thesis, Yale Univ. (1998)
19. Mantel, H., Sudbrock, H.: Flexible Scheduler-Independent Security. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 116–133. Springer, Heidelberg (2010)
20. Martin, W., White, P., Taylor, F.S., Goldberg, A.: Formal construction of the mathematically analyzed separation kernel. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, pp. 133–141 (2000)
21. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). The MIT Press (1997)
22. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991)
23. Nanevski, A.: A Modal Calculus for Exception Handling. In: Intuitionistic Modal Logics and Applications Workshop (IMLA 2005) (June 2005)
24. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis (1999)
25. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries, the Revised Report. Cambridge University Press (2003)
26. Russo, A., Sabelfeld, A.: Securing interaction between threads and the scheduler. In: Proc. of the 19th IEEE Workshop on Comp. Sec. Found., pp. 177–189 (2006)
27. Sabelfeld, A., Myers, A.C.: Language-based information flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
28. Swierstra, W., Altenkirch, T.: Beauty in the beast. In: Proceedings of the ACM SIGPLAN Haskell Workshop (Haskell 2007), pp. 25–36 (2007)
29. Wadler, P.: The marriage of effects and monads. In: Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming, pp. 63–74 (1998)

Verification of ATL Transformations Using Transformation Models and Model Finders

Fabian Büttner[1,⋆], Marina Egea[2,⋆⋆], Jordi Cabot[1], Martin Gogolla[3]

[1] AtlanMod Research Group, INRIA / Ecole des Mines de Nantes
[2] Atos Research & Innovation Dept., Madrid
[3] Database Systems Group, University of Bremen
fabian.buettner@inria.fr, marina.egea@atosresearch.eu,
jordi.cabot@inria.fr, gogolla@tzi.de

Abstract. In model-driven engineering, models constitute pivotal elements of the software to be built. If models are specified well, transformations can be employed for different purposes, e.g., to produce final code. However, it is important that models produced by a transformation from valid input models are valid, too, where validity refers to the metamodel constraints, often written in OCL. Transformation models are a way to describe this Hoare-style notion of partial correctness of model transformations using only metamodels and constraints. In this paper, we provide an automatic translation of declarative, rule-based ATL transformations into such transformation models, providing an intuitive and versatile encoding of ATL into OCL that can be used for the analysis of various properties of transformations. We furthermore show how existing model verifiers (satisfiability checkers) for OCL-annotated metamodels can be applied for the verification of the translated ATL transformations, providing evidence for the effectiveness of our approach in practice.

Keywords: Model transformation, Verification, ATL, OCL.

1 Introduction

In model-driven engineering (MDE), models constitute pivotal elements of the software to be built. Ideally, if these models are specified sufficiently well, model transformations can be employed for different purposes, e.g., they may be used to finally produce code. The increasingly popularity of MDE has led to a growing complexity in both models and transformations, and it is essential that transformations are correct if they are to play their key role. Otherwise, errors introduced by transformations will be propagated and may produce more errors in the subsequent MDE steps.

Our work focuses on checking partial correctness of declarative, rule-based transformations between constrained metamodels. More specifically, we consider the transformation language ATL [15] and metamodels in MOF [21] style (e.g., EMF [25], KM3 [16]) that employ OCL [20,27] constraints to precisely describe their domain.

⋆ This research was partially funded by the Nouvelles Eq́uipes program of the Pays de la Loire region (France).
⋆⋆ This research was partially funded by the EU project NESSoS (FP7 256890).

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 198–213, 2012.

These ingredients are popular due to their sophisticated tool support (in particular on the Eclipse platform) and because OCL is employed in almost all OMG specifications. Model transformations can be considered as programs that operate on instances of metamodels. In this sense, we can also apply the classical notion of correctness to model transformations. In this paper, we are interested in a Hoare-style notion of partial correctness, i.e., in the correctness of a transformation with respect to the constraints of the involved metamodels. In other words, we are interested in whether the output model produced by an ATL transformation is valid for any valid input model.

In this paper we present a verification approach based on *transformation models*. Transformation models are a specific kind of what is commonly called a 'trace model'. Given an ATL transformation $T : \mathcal{M}_I \rightarrow \mathcal{M}_F$ from a source metamodel $\mathcal{M}_I$ to a target metamodel[1] $\mathcal{M}_F$, a transformation model $\mathcal{M}_T$ is a metamodel that includes $\mathcal{M}_I$ and $\mathcal{M}_F$, and additional structural modeling elements and constraints in order to capture the execution semantics of T. In our opinion, this approach brings advantage because it reduces the problem of verifying rule-based transformations between constrained metamodels to the problem of verifying constrained metamodels only. This way, in terms of automated verification, we can reuse existing implementations and work for model verification, benefiting from the results achieved by a broad community over a decade.

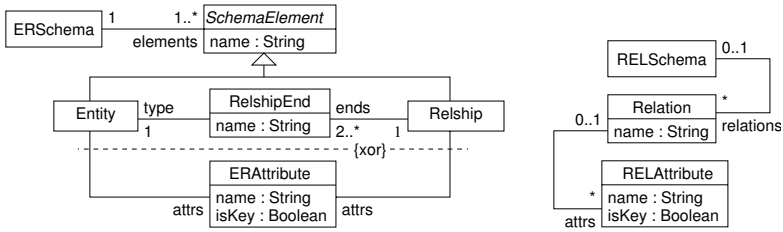
The transformation model methodology was first presented in [11] and [6]. We provided a first sketch of how to apply the methodology to ATL in [4]. In this paper, we now present a precise description of how to automatically generate transformation models from declarative ATL transformations. Furthermore, we show how existing model finders for OCL-annotated metamodels can be employed 'off-the-shelf' in practical verification. We employ a transformation ER-to-Relational (ER2REL) to illustrate our approach, as this example is well-known and conceptually 'dense' (it contains only few classes but comparatively many constraints). We show how the transformation model is derived using our algorithm and how it can be used to effectively verify the ATL transformation using UML2Alloy [1] and Alloy as a bounded model verification tool (with Alloy being based on SAT in turn). Notice, however, that the methodology is independent from a specific verification technique.

Organization. Sect. 2 describes the running example ER2REL. Sect. 3 shows how to derive transformation models for ATL. In Sect. 4 we present how UML2Alloy could be employed to validate ER2REL (based on the derived transformation model). Sect. 5 puts our contribution in the context of related work. We conclude in Sect. 6.

2 Running Example

We have chosen an ATL transformation (ER2REL) from a simple Entity-Relationship (ER) to a simple relational (REL) data model as a running example for our paper for two reasons. First, this domain is well-known (the results can be easily validated). Second, almost all elements are constrained by one or more invariants, including several

[1] For typographical reasons we use $\mathcal{M}_I$ ('initial') and $\mathcal{M}_F$ ('final') to denote the input and output.



(a) ER (b) REL

Fig. 1. ER and REL metamodels

```
context ERSchema inv ER_EN:                -- element names are unique in schema
  self.elements->forAll(e1,e2 |  e1.name=e2.name implies e1=e2)
context Entity inv ER_EAN:                 -- attr names are unique in entity
  self.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2)
context Relship inv ER_RAN:                -- attr names are unique in relship
  self.attrs->forAll(a1,a2 | a1.name = a2.name implies a1=a2)
context Entity inv ER_EK:                  -- entities have a key
  self.attrs->exists(a | a.isKey)
context Relship inv ER_RK:                 -- relships do not have a key
  not attrs->exists(a1 | a1.isKey)
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
context RELSchema inv REL_RN:              -- relation names are unique in schema
  relations->forall(r1,r2|   r1.name=r2.name implies r1=r2)
context Relation inv REL_AN:               -- attribute names  unique in relation
  self.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2)
context Relation inv REL_K:                -- relations have a key
  self.attrs->exists(a | a.isKey)
context RELSchema inv REL_mult1:  self.relations->size() > 0   -- mult. 1..*
context Relation inv REL_mult2:   self.schema <> null          -- mult. 1..1
context Relation inv REL_mult3:   self.relations->size() > 0   -- mult. 1..*
context RELAttribute inv REL_mult4: self.relation <> null      -- mult. 1..1
```

Fig. 2. OCL constraints for ER and REL

universal quantifiers. This makes the verification of this transformation reasonably hard.

Fig. 1 depicts the ER and REL metamodels[2]. Fig. 2 shows the corresponding OCL constraints. The constraints are as expected: names must be unique within their respective contexts, entities and relation must have a key, relationships must not have a key. Notice that we encoded the multiplicity constraints for REL as explicit OCL constraints (REL_mult). We only left the unrestricted multiplicities of $0..1$ (for object-typed navigations) and $0..*$ (for collection-typed navigations) in the class diagram, because we want to verify the validity of ER2REL w.r.t. these multiplicities later.

The ATL transformation ER2REL is shown in Fig. 3 and contains six rules: The first rule S2S maps ER schemas to REL schemas, the second rule E2R maps each entity to a relation, and the third rule R2R maps each relationship to a relation. The remaining

[2] Notice that we simply refer to the elements as entities, relationships, and relations instead of entity types, relationship types, and relation types.

```
module ER2REL; create OUT : REL from IN : ER;

rule S2S {
from s : ER!ERSchema
to   t : REL!RELSchema ( relations <- s.entities->union(s.relships) )}

rule E2R {
from s : ER!Entity
to   t : REL!Relation (name<-s.name, schema<-s.schema) }

rule R2R {
from s : ER!Relship
to   t : REL!Relation (name <-s.name, schema<-s.schema) }

rule EA2A {
from att : ER!ERAttribute, ent : ER!Entity (att.entity=ent)
to   t   : REL!RELAttribute (name<-att.name, isKey<-att.isKey, relation<-ent)}

rule RA2A {
from att : ER!ERAttribute, rs : ER!Relship (att.relship=rs)
to   t   : REL!RELAttribute (name<-att.name, isKey<-att.isKey, relation<-rs)}

rule RA2AK {
from att : ER!ERAttribute,
     rse : ER!RelshipEnd (att.entity=rse.entity and att.isKey=true)
to     t : REL!RELAttribute
                  (name<-att.name, isKey<-att.isKey, relation<-rse.relship)}
```

Fig. 3. Initial version of the ATL transformation ER2REL

three rules generate attributes for the relations. Both entity and relationship attributes are mapped to relation attributes (rules `EA2A` and `RA2A`). Furthermore, the key attributes of the participating entities are mapped to relation attributes as well (rule `RA2AK`).

All six rules of ER2REL are *matched rules*, which are the main constructs of ATL. A *matched rule* is composed of a source pattern and a target pattern. The source pattern specifies a set of objects of the source metamodel and uses, optionally, an OCL expression as a filtering condition. The target pattern specifies a set of objects of the target metamodel plus a set of bindings. The bindings describe assignments to features (i.e., attributes, references, and association ends) of the target objects. The execution semantics of matched rules can be described in three steps: First, the source patterns of all rules are matched against input model elements. Second, for every matched source pattern, the target pattern is followed to create objects in the target model. Notice that the execution of an ATL transformation always starts with an empty target model. In the third step, the bindings of the target patterns are executed. These bindings are performed straight-forwardly with one exception: If a value that is assigned to a property is an object of the input model, and if this object has been mapped by a rule in the previous step, then instead of the input object the (first) output object that has been created by this rule is used. By default, the ATL execution engine reports an error if no or multiple of such *matches* exist.

Next, in order to illustrate the ATL execution semantics, we explain how it works, for instance, for the rule `RA2A`. This rule is applied to every combination of an `ERAttribute att` and a `Relship rs` instance for which the condition `att.relship=rs` holds. For each such match, one `RELAttribute t` is created. The values of the `name` and `isKey` properties of `t` are simply copied from `att`. For the binding of the property relation, the implicit resolution strategy of ATL will replace the value of the input pattern element `rs` (which is an object of the source model) by a

reference to the Relation object that has been created by `R2R` for `rs`. In this case, `R2R` is the only rule that can be used to resolve `Relation`-objects. However, in general, there can be multiple rules for each type.

3 Transformation Models for ATL

Model transformations can be considered as programs that operate on instances of metamodels. In this sense, we can also apply notions of correctness for programs to model transformations. We will consider the input and output models of a transformation as valid if and only if they conform to the structure and to the constraints of their metamodels. Partial correctness then states that *if* the transformation produces an output model from a valid input model, that output model is valid as well. Total correctness extends this notion and states that the transformation produces a valid output for every valid input model (i.e., that the transformation terminates for every valid input model and does not abort with an error message).

Our notion of a transformation model $\mathcal{M}_T$ of a transformation $T : \mathcal{M}_I \to \mathcal{M}_F$ aims to support the verification of partial correctness of T using $\mathcal{M}_T$ as an equivalent surrogate as follows. A transformation model $\mathcal{M}_T$ is a metamodel (i.e., a structural specification of classes, associations, and constraints) that integrates $\mathcal{M}_I$ and $\mathcal{M}_F$ and additional structural modeling elements and constraints that capture the execution semantics of T. A pair of an $\mathcal{M}_I$ instance M_I and an $\mathcal{M}_F$ instance M_F is related by T if and only if there is an instance of $\mathcal{M}_T$, valid w.r.t. to all constraints, whose $\mathcal{M}_I$ part is M_I and whose $\mathcal{M}_F$ part is M_F. In practice, we want to loosen this equivalence to hold only for those M_I for which T terminates. However, for the declarative subset of ATL that we consider, recursive OCL helper operations are the only source of nontermination, as the actual execution of ATL rules is non-recursive and non-looping (and also deterministic [17]).

Having such an equivalent transformation model, we can verify partial correctness of T using 'off-the-shelf' model finders (e.g., based on SAT solving). In the remaining section, we show how to systematically derive such transformation models for ATL transformations. We provide a general algorithm for this (Sect. 3.1) and discuss the validity of our translation (Sect. 3.2).

3.1 An Algorithm to Derive Transformation Models for ATL

Our translation does cover a significant subset of ATL, namely *matched rules*, which are the workhorse of ATL, in the form provided in Fig. 4. We presume that all expressions and bindings in the transformation are correctly typed. We do not support imperative extensions, called or lazy rules at the moment, and we do not allow recursive OCL helper operations.

The algorithm that creates $\mathcal{M}_T$ for $T : \mathcal{M}_I \to \mathcal{M}_F$ is depicted in Fig. 5. It consists of four main steps. The results of the algorithm for ER2REL is shown in Fig. 6 (generated classes and associations) and Fig. 7 (generated constraints). The first step includes all elements (i.e., classes, associations, attributes, constraints) of $\mathcal{M}_I$ and $\mathcal{M}_F$.

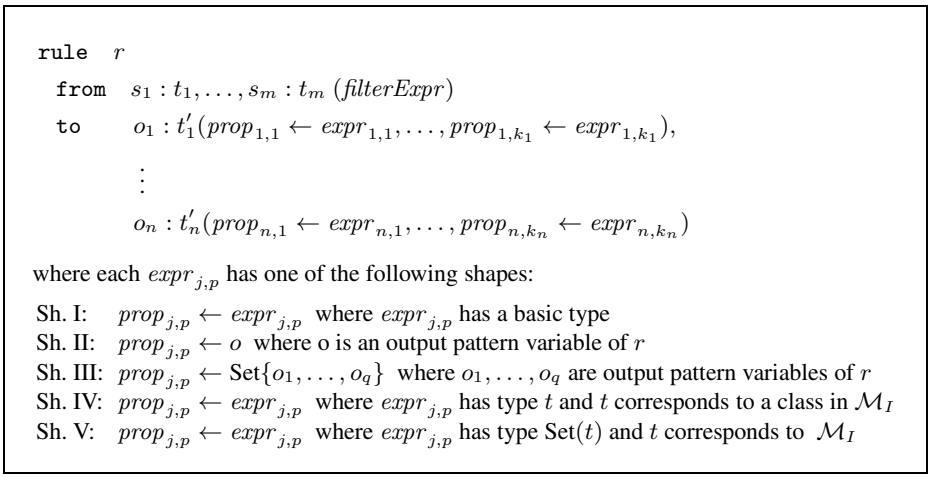


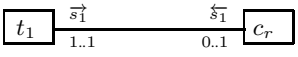
Fig. 4. ATL matched rule's patterns currently supported by our mapping

The second step adds a new class c_r for each rule r in T (step 1a; e.g., class 'S2S' in Fig. 6), connects c_r to the types of the input and output pattern variables (steps 1b and 1c). Notice that for rules with multiple pattern elements, the same input object can participate several times (with different partners), hence the '0..*' multiplicity. In the next step we add *matching constraints* that ensure that exactly those combinations of M_I objects are connected to a c_r object that are matched by r (steps 1d and 1e; e.g., match_EA2A and match_EA2A_cond in Fig. 7). For each binding to an output pattern object, corresponding *binding constraints* over c_r are added (step 1f; e.g., bind_E2R_t_name). For unassigned properties, a constraint is added that ensures that these properties are null (step 2g). The third step considers each class in $\mathcal{M}_F$ and adds a *creation constraint* to ensure that each M_F object is created by exactly one rule of T (e.g., create_Relation in Fig. 7). The fourth step is specific to those transformations that have potentially overlapping patterns. Recall that ATL does not allow a combination of M_I objects to be matched by more than one rule (the engine would abort in this case). The fourth step corresponding *mutual exclusion constraints* for all pairs of potentially overlapping rules (ER2REL does not contain such rules).

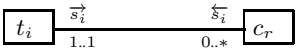
We make use of some auxiliary functions in the description of the algorithm that generate OCL expressions for the more complex constraints. We define them below. To create the associations that connect the classes c_r to the resp. class in $\mathcal{M}_I$ and $\mathcal{M}_F$, we assume $\overrightarrow{s}$ and $\overrightarrow{o}$ to name the the corresponding navigable association ends for the pattern variables s and o (from the perspective of the rule class), and $\overleftarrow{s}$ and $\overleftarrow{o}$ to generate unique opposite association end names (from the perspective of the resp. classes in $\mathcal{M}_I$ and $\mathcal{M}_F$). We use the hat notation $\hat{z}$ to denote a fresh variable.

Auxiliary function $matchExpr(r)$. The function $matchExpr(r)$ that we use in step 1d yields a Boolean OCL expression of m nested 'forAll' expressions for the m input

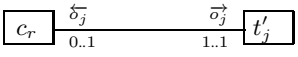
1. Copy all model elements of $\mathcal{M}_I$ and $\mathcal{M}_F$.
2. For each matched rule r in T let $s_1 : t_1, \ldots, s_m : t_m$ denote the input pattern variables of r and $o_1 : t'_1, \ldots, o_n : t'_n$ the output pattern variables of r. Then:
   (a) Add a class c_r.
   (b) If $m = 1$ (i.e., r has only single input pattern variable), add an association

   Else, if $m > 1$, add the following association for each $1 \leq i \leq m$

   (c) For each output pattern variable $o_j : t'_j$ of r with $1 \leq j \leq n$ add an association

   (d) Add a constraint **context** t_1 **inv** : $matchExpr(r)$.
   (e) Add a constraint **context** c_r **inv** : $filterExpr'$ where $filterExpr' = filterExpr[s_1 \ldots s_m]/[\text{self}.\overrightarrow{s_1} \ldots \text{self}.\overrightarrow{s_m}]$ is the filter expression with all input pattern variables are replaced by navigations from the rule object.
   (f) For each binding $prop_{j,p} \leftarrow expr_{j,p}$ to an output pattern variable o_j of r with $1 \leq j \leq n$ and $1 \leq p \leq k_n$, add a constraint **context** c_r **inv** : $\text{self}.\overrightarrow{o_j}.prop_{j,p} = resolve[\![\, expr'_{j,p} \,]\!]$ where $expr'_{j,p} = [s_1 \ldots s_m]/[\text{self}.\overrightarrow{s_1} \ldots \text{self}.\overrightarrow{s_m}]$.
   If $expr_{j,p}$ is of shape IV furthermore add a constraint **context** c_r **inv** : $(expr'_{j,p} = \text{null}) = (resolve[\![\, expr'_{j,p} \,]\!] = \text{null})$.
   If $expr_{j,p}$ is of shape V furthermore add a constraint **context** c_r **inv** : $expr'_{j,p} \rightarrow \text{size}() = resolve[\![\, expr'_{j,p} \,]\!] \rightarrow \text{size}()$.
   (g) For each property $prop$ of o_j that is not bound by r, we add a constraint **context** c_r **inv** : $\text{self}.\overrightarrow{o_j}.prop = \text{null}$.
3. For each class c in $\mathcal{M}_F$, if $\{o_1 : t'_1, \ldots, o_q : t'_q\} = creators(c)$ then add a constraint **context** c **inv** : $\text{self}.\overleftarrow{o_1} \rightarrow \text{size}() + \cdots + \text{self}.\overleftarrow{o_q} \rightarrow \text{size}() = 1$.
   Otherwise, when there are no creators for c, add a constraint **context** c **inv** : false.
4. For each pair of rules r, r' in T that have input patterns of the same size m and each sequence of $\mathcal{M}_I$ types $t''_1, \ldots, t''_m$, add a mutual exclusion constraint if r and r' potentially overlap on $t''_1, \ldots t''_m$:
   context t_1 **inv** : $mutexExpr(r, r', \langle t''_1, \ldots, t''_m \rangle)$
   The rules r and r' overlap on $t''_1, \ldots t''_m$ when $t''_i \leq t_i$ and $t''_i \leq t'_i$ holds for each i with $1 \leq i \leq m$.

Fig. 5. Algorithm

pattern elements of r such that for each combination of objects in M_I that matches r exactly one instance of c_r is connected to these objects. It is defined as follows.

$$\begin{aligned} matchExpr(r) := t_1 \rightarrow \text{forAll}(\hat{s}_1 \mid t_2 \rightarrow \text{forAll}(\hat{s}_2 \mid \ldots t_m \rightarrow \text{forAll}(\hat{s}_m \mid \\ filterExpr' \text{ implies } c_r.\text{allInstances}() \rightarrow \text{one}(\hat{z} \mid \\ \hat{z}.\overrightarrow{s_1} = \hat{s}_1 \text{ and } \ldots \text{ and } \hat{z}.\overrightarrow{s_m} = \hat{s}_m) \cdots) \end{aligned}$$

where $filterExpr' = filterExpr[s_1 \ldots s_m]/[\hat{s}_1 \ldots \hat{s}_m]$ is the filter expression of r in which the pattern variables are replaced by the variables used in the above iteration.

Auxiliary function $resolve[\![\, expr \,]\!]$. The function $resolve[\![\, expr \,]\!]$ that we use in step 1f is the most complex one. We use it to translate the implicit resolve mechanism of ATL into OCL. Recall that ATL, when processing a binding $prop \leftarrow expr$, replaces each object value from M_I by an object value from M_F. To do this, it uses the first output pattern variable of the (unary input pattern) rule that matched the resp. object in M_I. Let t be the type (for shape IV) resp. the element type (shape V) of $expr$. Let $\{(x_1 : t_1, y_1 : t'_1), \ldots, (x_q : t_q, y_q : t'_q)\}$ be the set of pairs $(x_i : t_i, y_i : t'_i)$ of the (only) input pattern variable and the first output pattern variable with $t \leq t_i$ or $t_i \leq t$, taken from all rules in T that have a unary input pattern (these are the rules that can map an object of type t). Notice that in this set we consider pattern variables of multiple rules in T.

- For shapes I, II, and III, no resolution is required, as the result is either a basic type or a (collection) value of M_F – recall that we have already replaced all target pattern variables o by self.$\overrightarrow{o}$ in step (2f). We have $resolve[\![\, expr \,]\!] := expr$.
- For shape IV we distinguish two cases. When we have $q = 1$ (there is only one rule that can possibly match this type), then we can translate the resolution into two simple navigation steps[3] (the type cast may be omitted when *expr* already has a sufficient specific type):

  $$resolve[\![\, expr \,]\!] := expr.\text{oclAsType}(t_1).\overleftarrow{x_1}.\overrightarrow{y_1}.$$

  When we have $q > 1$, then there are multiple potential rules to be used for this resolution step. Notice that there cannot be two rules applied at the same time (we guarantee this by mutual exclusion constraints), so there is at most one non-null element (M_F object) in the set expression below, and we can use the 'any' operator to deterministically select it.

  $$\begin{aligned} resolve[\![\, expr \,]\!] := \text{Set}\{&expr.\text{oclAsType}(t_1).\overleftarrow{x_1}.\overrightarrow{y_1}, \\ &\ldots, \\ &expr.\text{oclAsType}(t_q).\overleftarrow{x_q}.\overrightarrow{y_q}\} \rightarrow \text{any}(\hat{z} \mid \hat{z} <> \text{null}) \end{aligned}$$

- For shape V, the translation is similar to the previous one, but now we have to apply the resolution step to each element of the collection (using 'collect'). The intermediate result is a set that contains one bag (multi-set) for each rule that can

[3] Recall that only matched rules with unary input patterns are used, so $\overleftarrow{x_1}$ is an object-valued navigation, cf. step 2b.

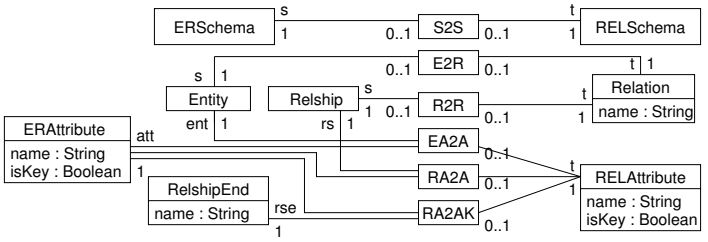


Fig. 6. Class diagram of the generated transformation model $\mathcal{M}_{\text{ER2REL}}$

potentially map t. We turn this into a flat set and remove the unmapped elements.

$$\begin{aligned} resolve[\![\, expr \,]\!] := \text{Set}\{\ & expr \rightarrow \text{collect}(\hat{z}|\hat{z}.\text{oclAsType}(t_1).\overleftarrow{x_1}.\overrightarrow{y_1}), \\ & \ldots, \\ & expr \rightarrow \text{collect}(\hat{z}|\hat{z}.\text{oclAsType}(t_q).\overleftarrow{x_q}.\overrightarrow{y_q})\} \\ & \rightarrow \text{flatten}() \rightarrow \text{select}(\hat{z}|\hat{z} <> \text{null}) \end{aligned}$$

Auxiliary function $creators(c)$. We use $creators(c)$ to identify all locations where a class of $\mathcal{M}_F$ may be instantiated. This is the set of all output pattern variables $\{o_1 : t_1, \ldots, o_q : t_q\}$ from the set of all rules of T with $t_j \leq c$ for each j with $1 \leq j \leq q$.

Auxiliary function $mutexExpr(r, r', \langle t''_1, \ldots, t''_n \rangle)$. This function yields a mutual exclusion expression for a pair of potentially overlapping rules r, r' in step 4. Recall that each tuple of M_I objects can be matched by at most one rule, otherwise the ATL engine aborts. Let $s_1 : t_1, \ldots s_m : t_m$ and $s'_1 : t'_1, \ldots s'_m : t'_m$ be the input pattern variables of the rules r and r'. Let $t''_1, \ldots, t''_m$ denote a sequence of $\mathcal{M}_I$ object types that can be matched potentially by both r and r'. The function $mutexExpr(r, r', \langle t''_1, \ldots, t''_n \rangle)$ generates an Boolean OCL expression that states that no combination of instances of $t''_1, \ldots t''_m$ can be connected to both a c_r and a $c_{r'}$ instance.

$$\begin{aligned} & mutexExpr(s, s', \langle t''_1, \ldots, t''_n \rangle) := \\ & \quad t''_1.\text{allInstances}() \rightarrow \text{forAll}(\hat{s_1}| \cdots t''_m.\text{allInstances}() \rightarrow \text{forAll}(\hat{s_m}| \\ & \qquad \text{not}(\, c_r.\text{allInstances}() \rightarrow \text{exists}(\hat{z}|\hat{z}.\overrightarrow{s_1} = \hat{s_1} \text{ and } \cdots \text{ and } \hat{z}.\overrightarrow{s_m} = \hat{s_m}) \text{ and} \\ & \qquad\quad c_{r'}.\text{allInstances}() \rightarrow \text{exists}(\hat{z'}|\hat{z'}.\overrightarrow{s'_1} = \hat{s_1} \text{ and } \cdots \text{ and } \hat{z'}.\overrightarrow{s'_m} = \hat{s_m}))\cdots) \end{aligned}$$

3.2 Validity of the Translation

As said in the beginning of this section, a transformation model $\mathcal{M}_T$ shall be equivalent to T (for transformations that do not employ recursive helper operations), in order to use $\mathcal{M}_T$ as a surrogate to verify the (partial) correctness of T. Recall that we defined the notion of a transformation model as follows: A pair of an $\mathcal{M}_I$ instance M_I and an $\mathcal{M}_F$ instance M_F is related by T *if and only if* there is an instance of $\mathcal{M}_T$ whose $\mathcal{M}_I$ part is M_I and whose $\mathcal{M}_F$ part is M_F. As there is not official formal semantics for ATL so far, we cannot prove formally that our axiomatization is correct. However, we justify

```
-- constraints generated by steps 2d and 2e: matching constraints
context ERSchema inv match_S2S:
  ERSchema.allInstances()->forAll(x1 : ERSchema |
    S2S.allInstances()->one(z : S2S | z.s = x1))

context Entity inv match_E2R:
  Entity.allInstances()->forAll(x1 : Entity |
    E2R.allInstances()->one(z : E2R | z.s = x1))

context Relship inv match_R2R:
  Relship.allInstances()->forAll(x1 : Relship |
    R2R.allInstances()->one(z : R2R | z.s = x1))

context ERAttribute inv match_EA2A:
  ERAttribute.allInstances()->forAll(x1 : ERAttribute |
    Entity.allInstances()->forAll(l_ent : Entity | x1.entity=(l_ent) implies
      EA2A.allInstances()->one(z : EA2A | z.att = x1 and z.ent = l_ent)))
context EA2A inv match_EA2A_cond: self.att.entity = self.ent

context ERAttribute inv match_RA2A:
  ERAttribute.allInstances()->forAll(x1 : ERAttribute |
    Relship.allInstances()->forAll(x2 : Relship | x1.relship=x2 implies
      RA2A.allInstances()->one(z : RA2A | z.att = x1 and z.rs = x2)))
context RA2A inv match_RA2A_cond: self.att.relship = self.rs

context ERAttribute inv match_RA2AK:
  ERAttribute.allInstances()->forAll(x1 : ERAttribute |
    RelshipEnd.allInstances()->forAll(x2 : RelshipEnd |
      x1.entity=x2.entity and x1.isKey implies
      RA2AK.allInstances()->one(z : RA2AK | z.att = x1 and z.rse = x2)))
context RA2AK inv match_RA2AK_cond: self.att.entity = self.rse.entity and
     self.att.isKey

-- constraints generated by step 2f: binding constraints
context S2S inv bind_S2S_t_relations:  self.t.relations =
    Set{self.s.elements->collect(z|z.oclAsType(Entity).e2r.t),
        self.s.elements->collect(z|z.oclAsType(Relship).r2r.t)}
    ->flatten()->select(z|z <> null)

context E2R inv bind_E2R_t_name: self.t.name = self.s.name
context R2R inv bind_R2R_t_name: self.t.name = self.s.name

context EA2A inv bind_EA2A_t_relation: self.t.relation = self.ent.e2r.t
context EA2A inv bind_EA2A_t_name: self.t.name = self.att.name
context EA2A inv bind_EA2A_t_isKey: self.t.isKey = self.att.isKey

context RA2A inv bind_RA2A_t_name: self.t.name = self.att.name
context RA2A inv bind_RA2A_t_relation: self.t.relation = self.rs.r2r.t
context RA2A inv bind_RA2A_t_isKey: self.t.isKey = self.att.isKey

context RA2AK inv bind_RA2AK_t_isKey: self.t.isKey = self.att.isKey
context RA2AK inv bind_RA2AK_t_relation: self.t.relation =
     self.rse.relship.r2r.t
context RA2AK inv bind_RA2AK_t_name: self.t.name = self.att.name

-- constraints generated by step 3: creation constraints
context RELSchema inv create_RELSchema: self.s2s->size() = 1
context Relation inv create_Relation: self.e2r->size() + self.r2r->size() = 1
context RELAttribute inv create_RELAttribute:
  self.ea2a->size() + self.ra2a->size() + self.ra2ak->size() = 1

-- no constraints generated by step 4 (mutual exclusion constraints)
```

Fig. 7. Constraints of the generated transformation model $\mathcal{M}_{\text{ER2REL}}$

our OCL axiomatization informally. In the following, we consider the different aspects of the execution semantics of ATL matched rules and give reasons why our translation into OCL constraints is appropriate. For the sake of brevity we simply say 'M_T over M_I and M_F' to state that M_T is an instance of $\mathcal{M}_T$ whose $\mathcal{M}_I$ part is M_I and whose $\mathcal{M}_F$ part is M_F.

Abnormal termination. For the considered subset of ATL (well-typed matched rules, no imperative extensions, no recursive helper operations), the engine will always halt, and there are only two abnormal terminations of applying a transformation T to an input model M_I. The first one is when two or more rules are applied to the same tuple of M_I objects. Our translation prevents this by mutual exclusion constraints (generated in step 4). The second one abnormal termination condition is when an M_I object cannot be resolved to an M_F object when processing the bindings. This condition is excluded by the constraints generated in step 2f. Thus, when T aborts on M_I, there is no instance of $\mathcal{M}_T$ that completes M_I.

Matching. The constraints generated in step 2d require that every tuple of objects that matches the input pattern of a rule r must be connected to exactly on instance of c_r. The 1..1 multiplicities generated for the input associations for c_r ensure that no other instances of c_r exist. Thus, taking also into account that $\mathcal{M}_T$ does exclude M_I instances that would result in abnormal termination on multiple matches, the matching constraints in $\mathcal{M}_T$ encode exactly the matching semantics of ATL.

Binding and Resolution. In ATL, an M_F object can only be created by one rule, and only by this rule the properties of that object are assigned. This is mirrored one-to-one by the binding constraints we generate in step 2f. We already justified that our auxiliary function $resolve$ encodes the implicit resolution mechanism of ATL. Thus, taking also into account that $\mathcal{M}_T$ does exclude M_I instances that would leave unresolved references, the binding constraints in $\mathcal{M}_T$ encode exactly the binding semantics of ATL.

Frame problem. So far, we have justified by the matching and binding constraints that an instance M_T over M_I and M_F exists *if* $M_F = T(M_I)$. The creation constraints created in step 3 guarantee that M_T does not contain any $\mathcal{M}_F$ objects that are not generated by a rule (as the transformation always starts with an empty output model). Furthermore, step 2g guarantees that properties are null unless they are assigned by a rule. Together, this concludes the *if and only if* correspondence regarding T and $\mathcal{M}_T$.

4 Employing Model Finders to Verify ATL Transformations

Having translated an ATL transformation T into a purely structural transformation model $\mathcal{M}_T$ (i.e., a metamodel consisting of classes and their properties, and constraints), we can employ 'off-the-shelf' model finders (model satisfiability checkers) to verify partial correctness of T w.r.t. the metamodel constraints of $\mathcal{M}_F$ using $\mathcal{M}_T$.

In particular, we can check whether T might turn a valid input model M_I into an invalid output model M_F as follows: Let con_i with $1 \leq i \leq n$ denote the i-th constraints

of $\mathcal{M}_F$. Let $\mathcal{M}_{\overline{F}_i}$ denote a modified version of $\mathcal{M}_F$ stripped of all its constraints and having one new constraint $negcon_i$ that is the negation of con_i. Let $\mathcal{M}_{\overline{T}_i}$ denote the transformation model constructed for $T : \mathcal{M}_I \rightarrow \mathcal{M}_{\overline{F}_i}$. T is correct w.r.t. con_i if and only if $\mathcal{M}_{\overline{T}_i}$ has no instance. If such an instance exist, its $\mathcal{M}_I$ is a counter example for which T produces an invalid result.

4.1 Verification Using UML2Alloy

We have implemented the presented translation as a so-called 'higher-order' ATL transformation, that is, an ATL transformation that takes an ATL transformation (the one to be verified, including the input and output metamodels) that produces the corresponding transformation model. The metamodels and constraints are technically represented using EMF and OCLinEcore. We employed the UML2Alloy model finder [1] to check the 'negated' transformation model (as explained before) for satisfiability.

UML2Alloy translates the metamodel and the OCL constraints into a specification for the Alloy tool, which implements bounded verification of relational logic. In the resulting specification, each class is represented as an Alloy signature each OCL constraint is represented by exactly one Alloy fact with the same name as the OCL constraint. Thus, we can check for the constraint subsumption easily by disabling and negating the facts (one after another) for the $\mathcal{M}_F$ constraints.

Table 1 shows the verification results for ER2REL. We verified all seven constraints of REL using an increasing number of objects per class (the maximum extent per signature must be specified when running Alloy). We can see that a counter example for (only) the constraints REL_AN can be found using at least three objects per class. This means that there exists a valid ER instance that is transformed into an invalid REL instance by ER2REL. Alloy presents the counter example in both an XML format and in a graphical, object-diagram like notation.

Figure 8 depicts such a counter example for REL_AN. Notice that the instances of the transformation model have a natural interpretation as a trace model of the original transformation, the counter example directly shows which objects are mapped by which rules. In Fig. 8, apparently, ER2REL does not treat reflexive relationships appropriately, while all attribute names are unique within their owning entities and relationships in the input model, the transformation generates identical attribute names within one relation in the output model. There are several ways to deal with this particular problem in ER2REL. As one solution we could modify rule RA2AK to use the name of the relationship end (instead of the key attribute) to determine the name of a foreign key attribute. But in this case, we must disallow combined keys, or we will get another violation of REL_AN in the next verification round. As a more general solution we could introduce qualified names for foreign keys (combining the name of the association end and the name of the key attribute). We leave it to the reader to decide what is the most appropriate solution for which situation. Instead we want to consider again Fig. 8 and emphasize the benefits of the counter examples that our method produces: The counter examples present at the same time the offending input model (that reveals the problem) and an explanation of the transformation execution (how the rules turn the input model into an invalid output model). In our view, this makes our method an intuitive and powerful tool for transformation developers.

Table 1. Avg. solving times (in seconds) using Alloy. Non-subsumed constraint marked with †. Undetected counter example marked by (*). All checks were conducted several times on a 2.2 Ghz office laptop running Alloy 4.1, Windows 7, and Java 7.

Obj/Class	Obj/Total	REL_RN	REL_AN†	REL_K	REL_M1	REL_M2	REL_M3	REL_M4
2	28	0.06	* 0.06	0.05	0.05	0.05	0.7	0.05
3	42	0.15	0.11	0.10	0.11	0.11	0.11	0.09
5	70	3.12	0.51	0.70	0.40	0.21	0.52	0.20
7	98	38.62	0.58	4.21	1.21	0.54	3.93	0.48
10	140	543.93	1.70	136.61	4.96	1.53	17.03	1.33

4.2 Scalability

Table 1 also provides some insights on the scalability of the verification method. Depending on the constraint, the verification time starts to become significant above 100 objects. Of course, these numbers are highly dependent on the constraint complexity. While the ER2REL example is simple in terms of the number of classes and associations, we consider it to have a comparatively high constraint complexity per class. We could confirm that larger class diagrams / larger instance sets do not necessarily increase the solving times, whereas harder (more overlapping, less tractable) constraints do. In this sense, we are confident that our method is applicable to larger metamodels as well. However, for the verification of industrial size metamodels and transformations, we expect that further heuristics and separation of concerns strategies will be required (e.g., metamodel pruning [23]).

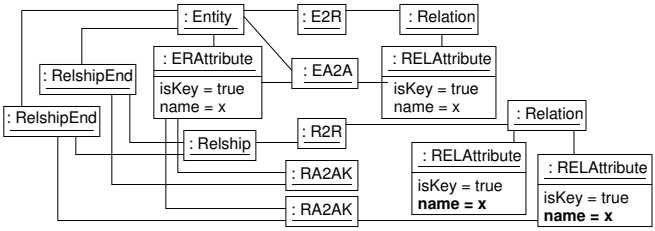


Fig. 8. Counter example: REL_AN violation

With respect to the chosen model verification tools (UML2Alloy and Alloy), it is important to remark again that these tools can only perform bounded verification. Thus, if Alloy cannot find a counter example, this does not mean that no counter example exists outside the fixed search bounds.

5 Related Work

We can relate our paper to several works. There are a couple of approaches that address partial, Hoare-style correctness of model transformation with respect to metamodel constraints as transformation pre- and postconditions. Inaba et al. automatically infer schema (i.e., metamodel) conformance for transformations based on the UnCAL

query language using the MONA solver [14]. The schema expressiveness in this approach is more restricted than OCL and describes only the typing of the graph. For example, uniqueness of names, as in ER and REL, could not be expressed. Asztalos et al. infer assertions for graph transformation-based model transformations [2]. They use an assertion language based on graph patterns, to enforce or avoid certain patterns in the model, which is a different paradigm than OCL. They provide explicit deduction rules for the verification (implemented in Prolog). On the contrary, we do not propose new deduction rules but rely on existing model finders. In similar vein, Rensink and Lucio et al. use model checking for the verification of first-order linear temporal [22] and pattern-based properties [19].

More specifically, there are also approaches that translate model transformations into transformation models in a similar fashion as we do: In a previous work, we translate triple graph grammars (which have a different execution semantics than ATL) and verify various conditions such weak and strong executability [7]. This work addresses executability but focuses on partial correctness (although we expect that executability could be expressed for ATL, too, using a tailored version of our algorithm). In similar vein Guerra et al. use triple graph grammar based transformation specifications and generate OCL invariants to check the satisfaction of these specifications by models [13]. To our knowledge, we are the only ones to present such a verification approach for ATL. Our paper is a successor of earlier results [4]. In that previous work, we gave a first sketch of the translation, but did not provide a complete algorithmic translation into OCL, as we do in our current contribution.

Related to the transformation model concept, the works of Braga et al., Cariou et al., and Gogolla and Vallecillo use OCL constraints to axiomatize properties of rule-based model transformation in terms of transformation contracts (but they do not generate them from a transformation specification as we do) [3,9,12].

To our knowledge, there are only two other approaches for the verification of ATL: First, Troya and Vallecillo provide a rewriting logic semantics for ATL and uses Maude to simulate and verify transformations, but do not consider the verification of Hoare style correctness [26]. Second, we recently presented an alternative approach to the formal verification of partial correctness of ATL using SMT solvers and a direct translation of the ATL transformation into first-order logic [5]. This approach is complementary to our current one and to other bounded verification approaches for ATL: It reasons symbolically and does not require bounds on the model extent, but it is incomplete (not all properties can be automatically decided this way, although it is refutationally complete in many cases). It can be used to verify several pre-post implications, but is not well suited to find counter examples. Furthermore, it builds on the translation of OCL to first-order logic by Egea and Clavel [10] which can only handle a subset of OCL. While the lightweight OCL axiomatization presented in our current work is fine for bounded model finders (and has an intuitive interpretation of counter examples as trace models), we were not able to employ SMT solvers for its verification. Using a direct translation of ATL+OCL into FOL [5] we could automatically prove several desired implications using the Z3 theorem prover solver (for the price that this approach requires of a full FOL encoding of ATL and OCL).

In this paper, we employed UML2Alloy [1] to perform the actual model verification. The community has developed several strong alternative approaches for the formal verification of models with constraints that we could use as well. They have in common that the model is translated into a formalism that has a well-defined semantics. Most approaches employ automated reasoning in the target formalism, for example, relational logic [18], constraint satisfaction problems [8], first-order logic [10], or propositional logic [24].

6 Conclusion and Future Work

In our paper, we have presented an approach that eases the verification of ATL transformations and thus helps to improve the quality of the MDE methodology in practice. As its core it is based on an automatic translation from ATL into a transformation model, which is a constrained metamodel that can be used as a surrogate for the verification of partial transformation correctness w.r.t. to the constraints of the input and output metamodels. We have presented a precise, executable description of the translation for a significant subset of ATL. We have also shown how this methodology can be implemented in practice using an ATL higher-order transformation and an 'off-the-shelf' model satisfiability checker (UML2Alloy). To our knowledge, we are the first ones to provide such an automatic approach for the verification of partial correctness for ATL.

We want to emphasize that the verification process can be automated as a "black box" technology, in the sense that the transformation developer is in contact only with models, in which the generated transformation models and their instances have a familiar representation for him.

In the future, we plan to explore the capabilities of different model finders as backends to our approach, in order to evaluate which are best suited for this kind of verification. Regarding ATL, we have already implemented an important subset of ATL, but we will incorporate (a restricted form) of so called *lazy rules*, which can be found in several transformations. Last but not least, comprehensive case studies must give more feedback on the applicability of our work.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MoDELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
2. Asztalos, M., Lengyel, L., Levendovszky, T.: Towards Automated, Formal Verification of Model Transformations. In: Proc. ICST 2010, pp. 15–24. IEEE Computer Society (2010)
3. Braga, C., Menezes, R., Comicio, T., Santos, C., Landim, E.: On the Specification, Verification and Implementation of Model Transformations with Transformation Contracts. In: Simao, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 108–123. Springer, Heidelberg (2011)
4. Büttner, F., Cabot, J., Gogolla, M.: On Validation of ATL Transformation Rules By Transformation Models. In: Proc. MoDeVVa 2011. ACM Digital Library (2012)
5. Büttner, F., Egea, M., Cabot, J.: On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 432–448. Springer, Heidelberg (2012)
6. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)

7. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)
8. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Proc. Automated Software Engineering, ASE 2007. ACM (2007)
9. Cariou, E., Belloir, N., Barbier, F., Djemam, N.: OCL contracts for the verification of model transformations. Electronic Communications of the EASST 24 (2009)
10. Clavel, M., Egea, M., de Dios, M.A.G.: Checking Unsatisfiability for OCL Constraints. Electronic Communications of the EASST 24, 1–13 (2009)
11. Gogolla, M.: Tales of ER and RE Syntax and Semantics. In: Transformation Techniques in Software Engineering. Dagstuhl Seminar Proc., vol. 05161. IBFI (2005)
12. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 221–235. Springer, Heidelberg (2011)
13. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: A Visual Specification Language for Model-to-Model Transformations. In: 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2010), pp. 119–126. IEEE Computer Society (2010)
14. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: Proc. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2011, pp. 17–28. ACM (2011)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
16. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
17. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
18. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
19. Lúcio, L., Barroca, B., Amaral, V.: A Technique for Automatic Validation of Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 136–150. Springer, Heidelberg (2010)
20. OMG: The Object Constraint Language Specification v. 2.2 (Document formal/2010-02-01). Object Management Group, Inc., Internet (2010), http://www.omg.org/spec/OCL/2.2/
21. OMG: Meta Object Facility (MOF) Core Specification 2.4.1 (Document formal/2011-08-07). Object Management Group, Inc., Internet (2011), http://www.omg.org
22. Rensink, A.: Explicit State Model Checking for Graph Grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 114–132. Springer, Heidelberg (2008)
23. Sen, S., Moha, N., Baudry, B., Jézéquel, J.-M.: Meta-model Pruning. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 32–46. Springer, Heidelberg (2009)
24. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
25. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Longman, Amsterdam (2008)
26. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. Journal of Object Technology 10, 5:1–5:29 (2011)
27. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley (2003)

Automatic Generation of Provably Correct Embedded Systems*

Shang-Wei Lin[1], Yang Liu[1,4], Pao-Ann Hsiung[2], Jun Sun[3], and Jin Song Dong[4]

[1] Temasek Laboratories, National University of Singapore
{tsllsw,tslliuya}@nus.edu.sg
[2] National Chung Cheng University, Chia-Yi, Taiwan
pahsiung@cs.ccu.edu.tw
[3] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[4] National University of Singapore
dongjs@comp.nus.edu.sg

Abstract. With the demand for new and complicated features, embedded systems are becoming more and more difficult to design and verify. Even if the design of a system is verified, how to guarantee the consistency between the design and its implementation remains a big issue. As a solution, we propose a framework that can help a system designer to model his or her embedded system using a high-level modeling language, verify the design of the system, and automatically generate executable software codes whose behavior semantics are consistent with that of the high-level model. We use two case studies to demonstrate the effectiveness of our framework.

1 Introduction

Embedded systems are increasingly used to control our daily life or critical tasks, which makes it important to guarantee the correctness. However, embedded systems are more and more complex due to the demand for new and complicated features, which makes it challenging to verify the correctness.

To verify embedded systems, system designers usually model the control behavior of the system using state-based diagrams such as UML state machines, where data and its operations are abstracted because modeling data and its operations by state-based diagrams is not straightforward. Furthermore, the number of possible system states often increases exponentially, which makes system verification (e.g., by model checking) infeasible. Even if data and its operations are specified by designers directly in high-level programming languages, the exact interactions among the user-written code and user-specified models are not precise, which leads to a possible semantic gap between the code and the models. During verification, if any fault occurs within this semantic gap, it will go undetected.

* This work is mainly supported by TRF Project "Research and Development in the Formal Verification of System Design and Implementation" from Temasek Lab@National University of Singapore; partially supported by project IDG31100105/IDD11100102 from Singapore University of Technology and Design, and project MOE2009-T2-1-072 from School of Computing@National University of Singapore.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 214–229, 2012.

In this work, we propose a framework as a solution to bridge the semantic gap. Our framework adopts *communication sequential program* (CSP#) [16] as our high-level modeling facility. CSP# is an expressive formal modeling language, which can be used to model concurrent processes communicating via both shared memory and message passing. CSP# even allows users to declare variables and model data operations on the declared variables. With its expressiveness, the control behavior and data operations of the system are formally and precisely modeled as a whole, which eliminates the semantic gap between control and data. In addition, to guarantee the consistency between the design of an embedded system and its implementation, the proposed framework automatically generates executable embedded software in a constructive way for the designer such that the functional correctness of the high-level CSP# models is transferred to the synthesized code. The contributions of this work include the following.

1. We propose a complete framework for modeling and verifying embedded systems.
2. The proposed framework can further automatically generate software code with functional correctness, and we have proved that the behavior semantics of the generated code conforms to the high-level model.

The rest of this paper is organized as follows. Section 2 gives related works. Section 3 gives an introduction to the CSP# language as well as its operational semantics. Section 4 introduces the proposed framework and proves the correctness of the generated code. We have applied our framework on two case studies, as given in Section 5. Section 6 concludes this paper and discusses some future works.

2 Related Work

Toolsets for design and verification of systems include the SCR toolset [3], NIMBUS [17], and SCADE Suite [15]. The SCR toolset uses the SPIN model checker, PVS-based TAME theorem prover, a property checker, and an invariant generator for formal verification of a real-time embedded system specified using the SCR tabular notation. It supports test case generation the TVEC toolset. NIMBUS is a specification-based prototyping framework for embedded safety-critical systems. It allows execution of software requirements expressed in RSML with various models of the environment, and it supports model checking and theorem proving. However, they do not support automatic code generation. SCADE Suite uses safe state machines (SSM) for requirement specification and automatically generates DO-178B Level A compliant and verified C/Ada code for avionics systems.

Worldwide research projects targeting embedded real-time software design include the MoBIES project [11] supported by USA DARPA, the HUGO project [7] supported by Germany's Ludwig-Maximilians-Universitä München, and the TIMES project [1] supported by the Uppsala University of Sweden. However, none of them completely support system designers to model, verify, and implement embedded systems with a comprehensive framework.

For automatic code generation, several works have been proposed on different formal models such as Objective-Z [13] and Event-B [10]. However, none of them proved the semantics consistency between the model and the generated code.

Verifiable Embedded Real-Time Application Framework (VERTAF) [4–6], is a comprehensive framework supporting high-level modeling, verification, and automatic code generation for real-time embedded systems. VERTAF adopts UML diagrams as its modeling language. Since UML diagrams are informal and have limited expressiveness, the exact interactions among the user-written high-level programming language code and the UML models cannot be precisely specified in VERTAF. This leads to a possible semantic gap between the user-written code and the user-specified models. During verification, if any fault occurs within this semantic gap, it will go undetected by the model checker. In this work, we extend the modeling ability of VERTAF with the CSP# language and enhance the ability of code synthesis such that executable verified software code can be generated automatically and the generated code is consistent with the high-level models.

3 Preliminaries

This section is devoted to a brief introduction to the CSP# language and its operational semantics. A CSP# *process* is defined (using a core subset of process constructs) as,

$$P ::= Stop \mid Skip \mid e\{prog\} \rightarrow P \mid ch!x \rightarrow P \mid ch?x \rightarrow P \mid P;Q \\ \mid [b]P \mid \text{if } b \ \{P\} \text{ else } \{Q\} \mid P \,|||\, Q \mid P[]Q$$

where e is an event with an operational sequential program $prog$, ch is a channel with a bounded buffer size, x is a variable, and b is a Boolean expression.

Let Σ denote the set of all visible events, τ denote an invisible action, and $\checkmark$ denote the the special event of termination. Let $\Sigma_\tau = \Sigma \cup \{\tau\}$ and $\Sigma_{\tau,\checkmark} = \Sigma \cup \{\tau, \checkmark\}$. The $Stop$ process communicates nothing (also called deadlock). The $Skip$ process is defined as $Skip = \checkmark \rightarrow Stop$. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a program $prog$ (also called *data operation*), the program is executed automatically together with the occurrence of the event. The attached program $prog$ can be C# program or any language with reflection that can be used for observing and/or modifying program execution at runtime. Channel communication process $ch!x \rightarrow P$ or $ch?x \rightarrow P$ behaves as P after sending or receiving x through the channel ch, respectively. Sequential composition $(P;Q)$ behaves as P until its termination and then behaves as Q. Conditional choice $if\ b\ \{P\}\ else\ \{Q\}$ behaves as P if b evaluates to true and behaves as Q otherwise. Process $[b]P$ waits until condition b becomes true and then behaves as P. Process $P \,|||^1 Q$ runs P and Q independently and they can communicate through shared variables. Process $P[]^2 Q$ behaves as either P or Q randomly. Example 1 illustrates how to model a system using CSP#.

Example 1. Peterson's algorithm [12] was designed for the synchronization problem for processes. Each process has to get into its critical section to do some computation,but only one process is allowed in its critical section at the same time. Listing 1.1

[1] In this work, we only consider the *interleaving* composition ($|||$) because *parallel* composition ($||$) is not natural and practical in real programs.

[2] CSP# provides *general*, *external*, and *internal* choice compositions. In this work, we focus on the general choice composition. The syntax and semantics of the three choice compositions can be found in PAT user manual.

Listing 1.1. CSP# Model for Peterson's Algorithm

```
var turn;        var pos[2];

P0() = req.0{pos[0] = 1; turn = 1} -> Wait0(); cs.0 -> reset.0{pos[0] = 0} -> P0();
Wait0() = if (pos[1] == 1 && turn == 1) { Wait0() } else { Stop };

P1() = req.1{pos[1] = 1; turn = 0} -> Wait1(); cs.1 -> reset.1{pos[1] = 0} -> P1();
Wait1() = if (pos[0] == 1 && turn == 0) { Wait1() } else { Stop };

Peterson() = P0() ||| P1();
```

shows the CSP# model of the Peterson's algorithm for two processes P_0 and P_1, where `req.0` and `cs.0` represent that P_0 requests to enter its critical section and P_0 is currently in its critical section, respectively.

Given a system modeled by a CSP# process, a *system configuration* is a three-tuple (P, V, C) where P is the current process expression, V is the current valuation of the global variables which is a mapping from a name to a value, and C is the current status of channels which is a mapping from a channel name to a sequence of items in the channel. A transition is of the form $(P, V, C) \xrightarrow{e} (P', V', C')$ meaning that (P, V, C) evolves to (P', V', C') by performing event e. The meaning or behavior of a CSP# process can be described by the operational semantics, as shown in Fig 1, where $e \in \Sigma$ and $e_\tau \in \Sigma_\tau$. We use $upd(V, prog)$ to denote the function which, given a sequential program $prog$ and V, returns the modified valuation function V' according to the semantics of the program. We use $V \models b$ (or $V \not\models b$) to denote that boolean condition b evaluates to true (or false) given V. We use $eva(V, exp)$ to denote the value of the expression evaluated with variable valuations in V. By the operational semantics, a CSP# process is associated with a labeled transition systems (LTS), as formulated in Definition 1.

Definition 1. ***(Labeled Transition System).*** *A* labeled transition system *(LTS) is a 4-tuple* $(S, \Sigma_{\tau,\checkmark}, \longrightarrow, s_0)$ *where* S *is a set of system configurations,* $\Sigma_{\tau,\checkmark}$ *is a set of events,* $\longrightarrow \subseteq S \times \Sigma_{\tau,\checkmark} \times S$ *is a transition relation, and* $s_0 \in S$ *is the initial state. We use* $s \xrightarrow{\alpha} s'$*, for simplicity, to denote* $(s, \alpha, s') \in \longrightarrow$ *where* $s, s' \in S$ *and* $\alpha \in \Sigma_{\tau,\checkmark}$*.*

4 Design and Synthesis Flow

Fig. 2 shows the overall flow of our approach. It consists of two main phases, namely, design phase and synthesis phase, and we refer to them as the front-end and back-end, respectively. The front-end is further divided into three subphases, namely, modeling, scheduling, and verification phases. There are two subphases in the back-end, namely, implementation mapping and code generation phases. The details of each phase are described in the following subsections.

4.1 Modeling

In the modeling phase, we adopt *communicating sequential programs* (CSP#) [16] as our modeling language, which is a high-level formal modeling language for concurrent

$$\frac{}{(Skip, V, C) \xrightarrow{\checkmark} (Stop, V, C)}\ [\text{skip}] \qquad \frac{(P, V, C) \xrightarrow{e} (P', V', C'), V \models b}{([b]P, V, C) \xrightarrow{e} (P, V, C)}\ [\text{guard}]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C)}{(P \,|||\, Q, V, C) \xrightarrow{e} (P' \,|||\, Q, V', C)}\ [\text{int1}] \qquad \frac{(Q, V, C) \xrightarrow{e} (Q', V', C)}{(P \,|||\, Q, V, C) \xrightarrow{e} (P \,|||\, Q', V', C)}\ [\text{int2}]$$

$$\frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C) \text{ and } (Q, V, C) \xrightarrow{\checkmark} (Q', V', C)}{(P \,|||\, Q, V, C) \xrightarrow{\checkmark} (P' \,|||\, Q', V, C)}\ [\text{int3}]$$

$$\frac{}{(e\{prog\} \rightarrow P, V, C) \xrightarrow{e} (P, upd(V, prog), C)}\ [\text{prog}]$$

$$\frac{(P, V, C) \xrightarrow{e} (P', V', C)}{(P\ ;\ Q, V, C) \xrightarrow{e} (P'\ ;\ Q, V', C)}\ [\text{seq1}] \qquad \frac{(P, V, C) \xrightarrow{\checkmark} (P', V', C)}{(P\ ;\ Q, V, C) \xrightarrow{\tau} (Q, V', C)}\ [\text{seq2}]$$

$$\frac{V \models b}{(if\ b\ \{P\}\ else\ \{Q\}, V, C) \xrightarrow{\tau} (P, V, C)}\ [\text{cond1}]$$

$$\frac{V \not\models b}{(if\ b\ \{P\}\ else\ \{Q\}, V, C) \xrightarrow{\tau} (Q, V, C)}\ [\text{cond2}]$$

$$\frac{(P, V, C) \xrightarrow{e_\tau} (P', V', C)}{(P\ []\ Q, V, C) \xrightarrow{e_\tau} (P', V', C)}\ [\text{ch1}] \qquad \frac{(Q, V, C) \xrightarrow{e_\tau} (Q', V', C)}{(P\ []\ Q, V, C) \xrightarrow{e_\tau} (Q', V', C)}\ [\text{ch2}]$$

$$\frac{C(ch) \text{ is not empty}}{(ch?x \rightarrow P, V, C) \xrightarrow{ch?C(ch).head} (P, V, C'), \text{where } C'(ch) = C(ch) \setminus \{C(ch).head\}}\ [\text{in}]$$

$$\frac{C(ch) \text{ is not full}}{(ch!exp \rightarrow P, V, C) \xrightarrow{ch!eva(V,exp)} (P, V, C'), \text{where } C'(ch) = C(ch) \cup \{eva(V, exp)\}}\ [\text{out}]$$

Fig. 1. Operational Semantics of CSP# Processes [16]

systems. In CSP#, complex communications such as shared memory and message passing can be easily modeled, and system designers can even include code fragments in the model. Most importantly, a complete simulation and verification tool support, *Process Analysis Toolkit* (PAT) [9], is available.

To model the interactions between system model and hardware, we classify hardware components and define generic hardware API for each class, e.g., for multi-media hardware components such as a LCD, we define `init()`, `reset()`, `display()`, and `refresh()`. Designers can use the generic hardware APIs as data operations in CSP# to describe the system behavior of interacting with hardware. Note that we assume hardware APIs have no side effects, i.e., they do not change the values of the declared variables in the model. Thus, they are not considered in the verification.

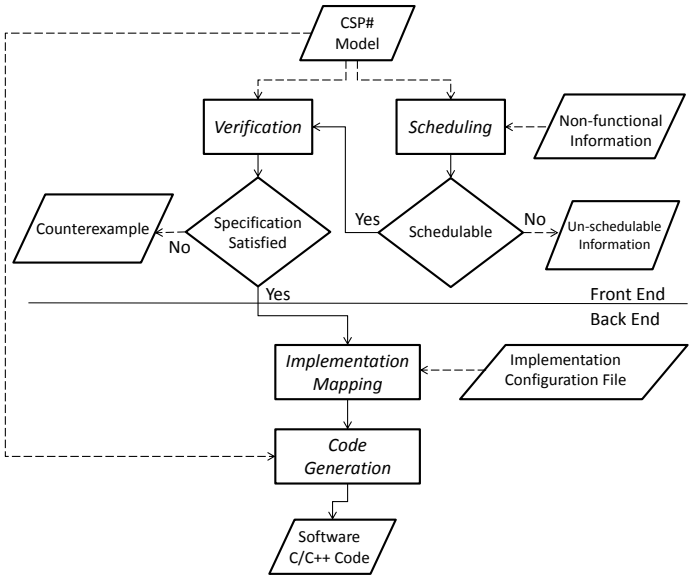


Fig. 2. Overall Flow

4.2 Scheduling

Non-functional requirements such as low-power consumption and worst-case execution time can be evaluated in this phase. To evaluate the non-functional parts of systems, the system designer can describe the power consumption or execution time of each event and each data operation (obtained by profiling) in the *non-functional information* file, and the requirements of non-functional properties are specified as assertions in CSP# in the modeling phase. Our framework provides several scheduling algorithms, such as *Quasi-dynamic Scheduling* (QDS) [6], to evaluate the non-functional properties. If the system design is not feasible, information on why it is non-schedulable will be given to designers. If the system design is schedulable, the flow goes to the verification phase.

4.3 Formal Verification

To check the functional correctness of the systems, we apply *model checking* [2] in this phase. Model checking is an automatic analysis procedure that can show if a system satisfies a temporal property or violates it with a counterexample. We integrate the PAT [9] model checker, which takes CSP# model as its input, with our framework. PAT is a self-contained model checker to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains. It implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. If the system model does not satisfy the

assertions of functional correctness, a counterexample will be given to the system designer. If the assertions are all satisfied, then the flow moves on to the back-end phase.

4.4 Implementation Mapping

To make the design of a system executable on a real hardware platform, every generic hardware API used in the modeling phase has to be mapped into a concrete implementation. In this phase, the target hardware platform where the final system runs is configured, and each generic hardware API is mapped into its corresponding implementation in the specified hardware platform. Hardware-dependent files such as make files and header files are all included in this phase.

4.5 Automatic Code Generation

In the code generation phase, executable software code is automatically generated from the high-level CSP# model in two steps. The CSP# processes are first translated into state machine models and then software code is synthesized from the translated state machines. The reasons for the two-phase synthesis instead of generating software code directing from CSP# model are as follows: (1) state machines are still the most popular models in industries, and the two-phase synthesis gives designers the flexibility to exchange state machine models of their system designs. (2) there is a mature middleware library, *Quantum Platform* (QP) [14], providing a programming paradigm for implementing the state machine models in C/C++ programming language. In the paradigm supported by QP, a state and a transition have their corresponding implementation patterns such that general principles can be concluded and code synthesis for state machines can be automated. In addition, QP supports many operating systems such as Linux/BSD, Windows/WinCE, μC/OS-II, etc., and many hardware platforms such as 80X86, ARM-Cortex/ARM9/ARM7, etc, which makes the generated software codes portable. Fig. 3 shows the architecture of QP.

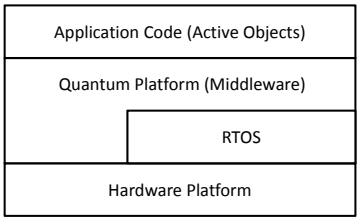


Fig. 3. Multi-Layer Code Generation

We formulate a state machine in Definition 2 and the interleaving, sequential, and choice compositions between two state machines in Definitions 3 to 5, respectively.

Definition 2. ***(State Machine)***. *A state machine is a* 6*-tuple* $\mathcal{M} = (S, \Sigma, B, A, s_0, T)$ *where* S *is a set of states;* Σ *is a set of events;* B *is a set of Boolean expressions;* A *is*

a set of actions; $s_0 \in S$ *is the initial state;* $T \subseteq S \times (\Sigma \cup \{\tau, \checkmark\}) \times B \times A^* \times S$ *is a transition relation. We use* $s \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} s'$ *to denote a transition, where* $s \in S$ *is the source state,* $s' \in S$ *is the destination state,* $e \in \Sigma$ *is the event trigger,* $b \in B$ *is the triggering condition, and* $(a_1; a_2; \cdots; a_n) \in A^*$ *is a sequence of actions for* $a_i \in A$ *and* $i \in \{1, 2, \ldots, n\}$*. We define* $Post(s, e) = \{s' \in S \mid s \xrightarrow{e/} s'\}$.

Definition 3. ***(Interleaving)****. Given two state machines* $\mathcal{M}_i = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *for* $i \in \{1, 2\}$*, the* interleave composition *is the state machine* $\mathcal{M}_1 \,|||\, \mathcal{M}_2 = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, B_1 \cup B_2, A_1 \cup A_2, s_0^1 \times s_0^2, T)$ *where* T *is defined as follows:*

$$
\begin{array}{ll}
(s_1, s_2) \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} (s_1', s_2) & \text{if } s_1 \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} s_1' \\
(s_1, s_2) \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} (s_1, s_2') & \text{if } s_2 \xrightarrow{e[b]/a_1;a_2;\cdots;a_n} s_2' \\
(s_1, s_2) \xrightarrow{\checkmark/} (s_1', s_2') & \text{if } s_1 \xrightarrow{\checkmark/} s_1' \text{ and } s_2 \xrightarrow{\checkmark/} s_2'
\end{array}
$$

Definition 4. ***(Sequential Composition)****.*
Given two state machines $\mathcal{M}_i = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *for* $i \in \{1, 2\}$*, the* sequential composition *is the state machine* $(\mathcal{M}_1; \mathcal{M}_2) = (S_1 \cup S_2, \Sigma_1 \cup \Sigma_2, B_1 \cup B_2, A_1 \cup A_2, s_0^1, T)$ *where* T *is defined as follows, where* $s \in S_1$*.*

$$T = T_1 \cup T_2 \setminus \{s \xrightarrow{\checkmark/} s' \in T_1 \mid s' \in Post(s, \checkmark)\} \cup \{s \xrightarrow{\tau/} s_0^2 \mid Post(s, \checkmark) \neq \emptyset\}.$$

Definition 5. ***(Choice Composition)****.*
Given two state machines $\mathcal{M}_i = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *for* $i \in \{1, 2\}$*, the* choice composition *is the state machine* $(\mathcal{M}_1 [] \mathcal{M}_2) = (S_i, \Sigma_i, B_i, A_i, s_0^i, T_i)$ *where* i *is randomly chosen from* $\{1, 2\}$*.*

Fig. 4 shows the one-to-one mapping from CSP# processes into state machines. The translation is performed constructively according the mapping step by step for each CSP# process. The translated state machine $\mathcal{M}_0$ for process P_0 in Example 1 is shown in Fig. 5. We omit the translated state machine $\mathcal{M}_1$ for process P_1 here since it is the same as M_0 except the conditions on transitions are symmetric to those of $\mathcal{M}_0$.

Theorem 1 proves that the behavior of the translated state machines is consistent with the behavior of the original CSP# models based on the concept of bisimulation.

Definition 6. ***(Bisimulation)****. Given two LTS* $L_i = (S_i, \Sigma, \longrightarrow_i, s_0^i)$ *for* $i \in \{1, 2\}$*, we say two states* $p \in S_1$ *and* $q \in S_2$ *are* bisimulation *of each other, denoted by* $p \approx q$ *iff*

- *for all* $\alpha \in \Sigma$ *if* $p \xrightarrow{\alpha}_1 p'$*, then there exists* q' *such that* $q \xrightarrow{\alpha}_2 q'$ *and* $p' \approx q'$
- *for all* $\alpha \in \Sigma$ *if* $q \xrightarrow{\alpha}_2 q'$*, then there exists* p' *such that* $p \xrightarrow{\alpha}_1 p'$ *and* $p' \approx q'$

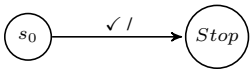
We say L_1 *and* L_2 *are* bisimulation *of each other, denoted by* $L_1 \approx L_2$ *iff* $s_0^1 \approx s_0^2$*.*

Theorem 1. *The translated state machine is a bisimulation of the original CSP# model.*

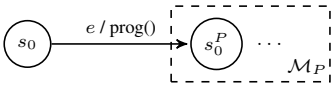
Proof. Given a CSP# process expression $\mathcal{E}$, let the labeled transition system associated with the process be $L_{\mathcal{E}} = (O, \Sigma_{\tau,\checkmark}, \longrightarrow_1, o_0)$. Let the translated state machine w.r.t. the CSP# process be $\mathcal{M}_{\mathcal{E}}$ and its associated labeled transition system be $L_{\mathcal{M}_{\mathcal{E}}} = (S, \Sigma_{\tau,\checkmark}, \longrightarrow_2, s_0)$. We want to prove that $L_{\mathcal{E}} \approx L_{\mathcal{M}_{\mathcal{E}}}$, i.e., $o_0 \approx s_0$. We use $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$ to denote $L_{\mathcal{E}} \approx L_{\mathcal{M}_{\mathcal{E}}}$. It can be proved by a structural induction on the CSP# expression from the following primitive processes.

(a) $Stop$

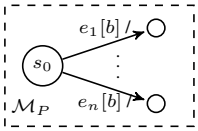
(b) $Skip$



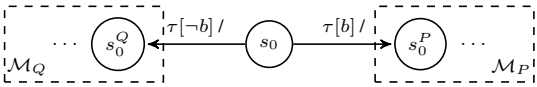
(c): $e\{prog\} \rightarrow P$



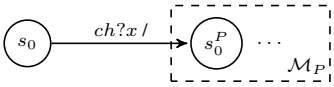
(d) $[b]P$



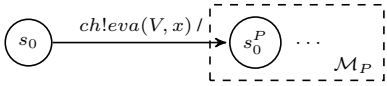
(e) $if\ b\ \{P\}\ else\ \{Q\}$



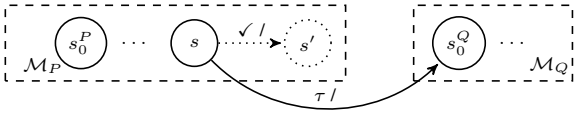
(f) $ch?x \rightarrow P$



(g) $ch!x \rightarrow P$



(h) $P;Q$



(h) $P[]Q$

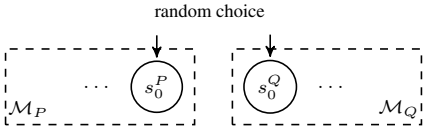


Fig. 4. Translation Rules from CSP# Processes and State Machines

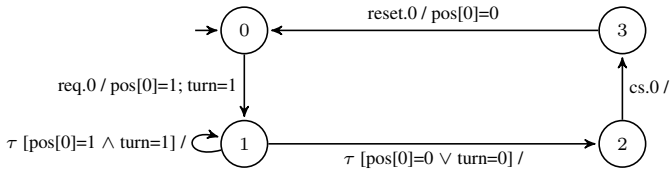


Fig. 5. Generated State Machine $\mathcal{M}_0$ for Process P_0

- $Stop$: By CSP# operational semantics, there is no transition rule for the $Skip$ process, so L_{Stop} is a LTS having a single state without any transitions. Its corresponding state machine $\mathcal{M}_{Stop}$ also has one state without any transitions, as shown in Fig. 4 (a). Thus, $Stop \approx \mathcal{M}_{Stop}$.
- $Skip$: By CSP# operational semantics, $Skip = \checkmark \rightarrow Stop$. the LTS $L_{\mathcal{E}}$ has two states o_0, o and one transition $o_0 \xrightarrow{\checkmark} o$, where $o_0 = (Skip, V, C)$ and $o = (Stop, V, C)$. The translated state machine $\mathcal{M}_{\mathcal{E}}$ has two states s_0, s and one transition $s_0 \xrightarrow{\checkmark/} s$, as shown in Fig. 4 (b). It is obvious $o_0 \approx s_0$. Thus, $Skip \approx \mathcal{M}_{Skip}$.
- $\mathcal{E} = e\{prog\} \rightarrow P$: By CSP# operational semantics, the LTS $L_{\mathcal{E}}$ takes transition $o_0 \xrightarrow{e} o$ and behaves like P, where $o_0 = (e\{prog\} \rightarrow P, V, C)$ and $o = (P, upd(V, prog), C)$. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. We add transition $s_0 \xrightarrow{e/prog()} s_0^P$ in the translated state machine $\mathcal{M}_{\mathcal{E}}$, as shown in Fig. 4 (c), where s_0^P is the initial state of $\mathcal{M}_P$. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = [b]P$: By CSP# operational semantics, process $[b]P$ behaves like P only if b holds, i.e., process P can perform an event only if b holds. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. For each outgoing transition from the initial state of $\mathcal{M}_P$, We add a triggering condition b, as shown in Fig. 4 (d), which guarantees that each outgoing transition from the initial state is taken only if b holds. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = if\ b\ \{P\}\ else\ \{Q\}$: By CSP# operational semantics, the LTS $L_{\mathcal{E}}$ may take transition $o_0 \xrightarrow{\tau} o'$ if b holds; otherwise it takes transition $o_0 \xrightarrow{\tau} o''$, where $o_0 = (if\ b\ \{P\}\ else\ \{Q\}, V, C)$, $o' = (P, V, C)$, and $o'' = (Q, V, C)$. Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the translated state machines w.r.t. P and Q, respectively, and $P \approx \mathcal{M}_P$, $Q \approx \mathcal{M}_Q$. In the initial state s_0 of $\mathcal{M}_{\mathcal{E}}$, two outgoing transitions $s_0 \xrightarrow{\tau[b]/} s_0^P$ and $s_0 \xrightarrow{\tau[\neg b]/} s_0^Q$ are available, as shown in Fig. 4 (e), where s_0^P and s_0^Q are the initial states of $\mathcal{M}_P$ and $\mathcal{M}_Q$, respectively. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = ch?x \rightarrow P$: By CSP# operational semantics, if the channel ch is not empty, the LTS $L_{\mathcal{E}}$ takes transition $o_0 \xrightarrow{ch?C(ch).head} o$, where $o_0 = (ch?x \rightarrow P, V, C)$, $o = (P, V, C')$, and $C'(ch) = C(ch) \backslash \{C(ch).head\}$, and behaves like P. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. In the initial state s_0 of $\mathcal{M}_{\mathcal{E}}$, we add transitions $s_0 \xrightarrow{ch?C(ch).head/} s_0^P$, as shown in Fig. 4 (f), where s_0^P is the initial state of $\mathcal{M}_P$. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.

- $\mathcal{E} = ch!x \rightarrow P$: By CSP# operational semantics, if the channel ch is not full, the LTS $L_{\mathcal{E}}$ takes transition $o_0 \xrightarrow{ch!eva(V,x)} o$, where $o_0 = (ch!x \rightarrow P, V, C)$, $o = (P, V, C')$, and $C'(ch) = C(ch) \cup \{eva(V, exp)\}$, and then behaves like P. Let $\mathcal{M}_P$ be the translated state machine w.r.t. process P such that $P \approx \mathcal{M}_P$. In the initial state s_0 of $\mathcal{M}_{\mathcal{E}}$, we add transitions $s_0 \xrightarrow{ch!eva(V,x)/} s_0^P$, as shown in Fig. 4 (g), where s_0^P is the initial state of $\mathcal{M}_P$. It is obvious that $o_0 \approx s_0$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = P \,|||\, Q$: By CSP# operational semantics, the transitions available in $L_{\mathcal{E}}$ are

$$
\begin{array}{l}
(P \,|||\, Q, V, C) \xrightarrow{e} (P' \,|||\, Q, V', C) \quad \text{if } (P, V, C) \xrightarrow{e} (P', V', C), e \neq \checkmark \\
(P \,|||\, Q, V, C) \xrightarrow{e} (P \,|||\, Q', V', C) \quad \text{if } (Q, V, C) \xrightarrow{e} (Q', V', C), e \neq \checkmark \\
(P \,|||\, Q, V, C) \xrightarrow{\checkmark} (P' \,|||\, Q', V', C) \quad \text{if } \begin{cases} (Q, V, C) \xrightarrow{\checkmark} (Q', V', C) \\ (P, V, C) \xrightarrow{\checkmark} (P', V, C) \end{cases}
\end{array}
$$

  Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the two translated state machines w.r.t. P and Q, respectively, such that $P \approx \mathcal{M}_P$ and $Q \approx \mathcal{M}_Q$. The state machine $\mathcal{M}_{\mathcal{E}}$ w.r.t. $\mathcal{E}$ is defined as $\mathcal{M}_{\mathcal{E}} = \mathcal{M}_P \,|||\, \mathcal{M}_Q$. By Definition 3, the only three transitions in $\mathcal{M}_{\mathcal{E}}$ correspond to the above three transitions in $L_{\mathcal{E}}$, respectively. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = P; Q$: By CSP# operational semantics, process $\mathcal{E}$ first behaves like P until P's termination and then behaves like Q. Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the two translated state machines w.r.t. P and Q, respectively, such that $P \approx \mathcal{M}_P$ and $Q \approx \mathcal{M}_Q$. The translated state machine $\mathcal{M}_{\mathcal{E}}$, as shown in Fig. 4 (f), first behaves like $\mathcal{M}_P$. Right before the termination of $\mathcal{M}_P$, it takes transition $s \xrightarrow{\tau/} s_0^Q$ and then behaves like $\mathcal{M}_Q$, which corresponds to the operation semantics of $\mathcal{E}$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.
- $\mathcal{E} = P[]Q$: By CSP# operational semantics, process $\mathcal{E}$ behaves like either P or Q randomly. Let $\mathcal{M}_P$ and $\mathcal{M}_Q$ be the two translated state machines w.r.t. P and Q, respectively, such that $P \approx \mathcal{M}_P$ and $Q \approx \mathcal{M}_Q$. The translated state machine $\mathcal{M}_{\mathcal{E}}$, as shown in Fig. 4 (i), behaves like either $\mathcal{M}_P$ or $\mathcal{M}_Q$ randomly, which corresponds to the operation semantics of $\mathcal{E}$. Thus, $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$.

Since any CSP# process expression $\mathcal{E}$ is composed with primitive processes inductively and we have proved that for each primitive process, the translated state machine is a bisimulation of it, therefore we can conclude that $\mathcal{E} \approx \mathcal{M}_{\mathcal{E}}$ for any CSP# process. □

The executable software code is then synthesized from the translated state machine models. Let us take the Peterson's algorithm in Example 1 to illustrate how software code is automatically generated. In QP, an active object consists of a logical state machine structure, an event queue, and an execution thread, as shown in Fig. 6. QP provides a base class, `QActive`, to implement an active object. A state machine is implemented as a subclass of the `QActive` class, and each state is implemented as a member function of the class. Listing 1.2 shows the declaration file of state machine $\mathcal{M}_0$ in Fig. 5, where the four states 0, 1, 2, 3 are implemented as four member functions `P0_0()`, `P0_1()`, `P0_2()`, and `P0_3()`, respectively. Each execution thread of an active object is responsible for dispatching events in its event queue, i.e., invoking the member

Listing 1.2. P0.h

```
1  class P0 : public QActive {
2     private: static QState initial(P0 *me, QEvent const *e);
3              static QState P0_0(P0 *me, QEvent const *e);
4              static QState P0_1(P0 *me, QEvent const *e);
5              static QState P0_2(P0 *me, QEvent const *e);
6              static QState P0_3(P0 *me, QEvent const *e);
7              QTimeEvt m_timeEvt;
8     public:  P0():QActive((QStateHandler)&P0::initial),m_timeEvt(TIMEOUT_SIG){};
9              ~P0(){};
10             void req_0()        {       std::cout<<"req_0"<<std::endl;   }
11             void cs_0()         {       std::cout<<"cs_0"<<std::endl;    }
12             void reset_0()      {       std::cout<<"reset_0"<<std::endl;         }
13 };
```

function representing the current state and passing the event as the argument. A transition from state s to s' in the state machine is implemented by invoking the `Q_TRAN(`s'`)` macro provided by QP.

An active object can communicate with others via shared variables and message passing. QP provides a *publish-subscription* mechanism for supporting message passing communication among active objects. Once an active object publishes an event, QP delivers the event to the event queue of whoever subscribes it, and each subscriber will receive the event by the execution thread. The dotted arrows in Fig. 6 show the paths of event passing between active objects and QP.

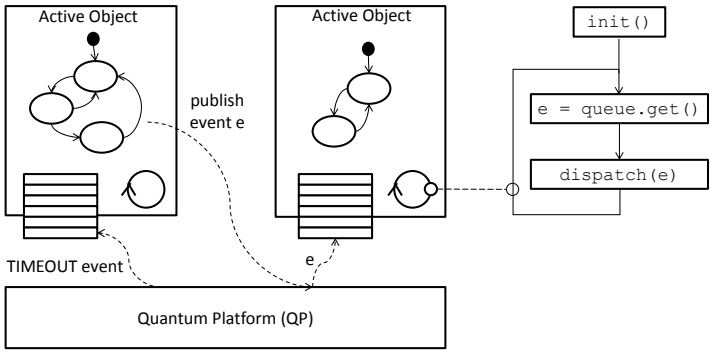


Fig. 6. Communications among Active Objects

QP also provides the timer facility such that an active object can register a timeout event of a certain time interval. After the time interval since the timer is fired, QP puts a timeout event into the event queue of the active object that registered it. Let us recall $\mathcal{M}_0$ in Fig. 5, we implement state 0 by registering a timeout event (Line 6 in Listing 1.3). After the timeout event occurs, the active object performs the `req_0()` function representing the CSP# event `req.0` and assigns value 1 to both of the variables

Listing 1.3. P0.cpp

```
1  QState P0::initial(P0 *me, QEvent const *){      return Q_TRAN(&P0::P0_0);      }
2
3  QState P0::P0_0(P0 *me, QEvent const *e){
4     switch(e->sig){
5        case Q_ENTRY_SIG:
6           me->m_timeEvt.postIn(me, WAIT_TIME);    return Q_HANDLED();
7        case TIMEOUT_SIG:
8           me->req_0(); pos[0]=1; turn = 1;
9           return Q_TRAN(&P0::P0_1);
10    }  return Q_SUPER(&QHsm::top);
11 }
12
13 QState P0::P0_1(P0 *me, QEvent const *e){
14    switch(e->sig){
15       case Q_ENTRY_SIG:
16          me->m_timeEvt.postIn(me, WAIT_TIME);    return Q_HANDLED();
17       case TIMEOUT_SIG:
18          if(pos[1] == 1 && turn == 1) { return Q_TRAN(&P0::P0_1); }
19          else { return Q_TRAN(&P0::P0_2); }
20    }  return Q_SUPER(&QHsm::top);
21 }
22                ...
```

`pos[0]` and `turn`, which corresponds to Line 8 in Listing 1.3. Then it transits to state 1 by invoking the `Q_TRAN` macro provided by QP (Line 9).
Theorem 2 proves that the behavior of implementation in active objects conforms to the behavior of the state machines.

Theorem 2. *The behavior of the implementation conforms to the state machine.*

Proof. We give our proof sketch here. To prove that for each state machine, the behavior of its implementation in active object conforms to the original one, let us recall and analyze what the execution thread of each active object does, as shown at the right side of Fig 6. The execution thread keeps doing the followings: if there is an event in the event queue, it removes the event and dispatches the event by invoking the member function representing the current state and passing the event as the argument. In the current active state, the pointer to the member function representing the current active state is changed to its successor state by invoking `Q_TRAN()` macro provided by QP. For each transition $A \xrightarrow{e[b]/act()} B$ from state A to state B, the call graph for QP functions and member functions representing states A and B is as shown in Fig. 7, which satisfies the operational semantics of state machines. Since the implementation for each transition satisfies the operational semantics of state machines, we can conclude that the implementation for the whole active object conforms to the original state machine.

In CSP# operational semantics, the execution of each transition is atomic. If we want to conclude that two-phase code generation is sound by Theorems 1 and 2, we have to make an assumption that the execution of the action on a transition of a state machine is also atomic. Fortunately, this can be guaranteed by taking and releasing a mutex at the beginning and the end of the action, respectively, which is exactly how we implement in our framework. Thus, the code generation in our framework is sound. □

Fig. 7. Call Graph of A State Transition

Listing 1.4. CSP# Model for the Entrance Guard System

```
1  #define RESET 77;
2  var pin[4];      var result = 0;          var open = 0;
3  var alarm = 0;   var symbol = 0;
4  channel pw 0;    channel i2ctr 0;         channel ctr2db 0;       channel db2ctr 0;
5  channel i2d 0;   channel ctr2a 0;         channel ctr2alm 0;      channel flag 0;
6
7  User() = User2()[]User1(); flag?x-> if(open == 1) { enter-> Skip } else { User() };
8  User1() = pw!1 -> pw!1 -> pw!1 -> pw!1 -> Skip;
9  User2() = pw!2 -> pw!RESET -> pw!2 -> pw!2 -> pw!2 -> pw!2 -> Skip;
10 Input() = pw?x -> check{symbol = x} -> if(symbol == RESET) { Input() } else {
11    input1 { pin[0] = symbol } -> i2d!symbol -> pw?y -> check{ symbol = y } ->
12    if(symbol == RESET) { Input() } else {
13       input2{ pin[1] = symbol } -> i2d!symbol -> pw?z ->check{ symbol = z } ->
14       if(symbol == RESET){ Input() } else {
15          input3 { pin[2] = symbol } -> i2d!symbol-> pw?k -> check{symbol = k} ->
16          if(symbol == RESET) { Input() } else{ input4{ pin[3] = symbol } ->
17          i2d!symbol -> i2ctr!1 -> Input() }
18       }
19    }   };
20 Display() = i2d?x -> show { display('*') } -> Display();
21 Controller() = i2ctr?x -> ctr2db!x -> db2ctr?x ->
22    if(result ==1) { ctr2a!1 -> flag!1 -> Controller() }
23    else { ctr2alm!1 -> flag!1 -> Controller() };
24 DBMS() = ctr2db?x ->
25    if(pin[0]==1 && pin[1]==1 && pin[2]==1 && pin[3]==1) {
26       checkOK{result = 1;}->  db2ctr!1 -> DBMS()
27    } else { checkFail{result = 0;} -> db2ctr!1 -> DBMS() };
28 Alarm() = ctr2alm?x -> alarmon{alarm = 1} -> alarmoff{alarm = 0} -> Alarm();
29 Actuator() = ctr2a?x -> opendoor{open = 1;} -> closedoor{open = 0} -> Actuator();
30 System() = User() ||| Input() ||| Controller() ||| DBMS() |||
31           Display() ||| Actuator() ||| Alarm();
32 #define pre pin[0]==1 && pin[1]==1 && pin[2]==1 && pin[3]==1;
33 #assert System() |= [] pre -> <> open == 1;
34 #assert System() reaches (result == 0 && open == 1);
35 #assert System() |= [] result == 0 -> <> alarm == 1;
```

5 Case Studies

We have applied the proposed framework on two case studies, namely an entrance guard system and a secure communication box. Both of the systems are modeled using the CSP# language and verified by the PAT model checker, and executable software codes for both systems are generated by our framework automatically.

The entrance guard system (EGS) controls the entrance of a building and it consists of six components, namely Input, Display, Controller, DBMS, Actuator, and Alarm, which are modeled as six CSP# processes in Listing 1.4. Input is a keypad receiving the 4-digit password as user input. Once a user enters a digit, it saves the digit to the `PIN` array and sends the digit to Display via the channel `i2d`. If the user presses the reset button, Input collects the 4-digit password from the first digit. After receiving four digits, it sends the password to Controller via the channel `i2ctr` (Lines 10-19). Display receives digits from Input and prints a '*' symbol by calling the hardware API `display()` for each digit on the LCD (Line 20). Controller sends the query

Listing 1.5. CSP# Model for the Secure Communication Box

```
1 #define UserConnect 1;          #define Data 2;          #define UserDisconnect 3;
2 channel network 0;              var packet;
3
4 User() = network!UserConnect -> network!Data -> network!UserDisconnect -> User();
5 Box() = poweron -> init -> network?UserConnect -> Connected(); Box();
6 Connected() = network?x -> store{packet=x} ->
7               if(packet == Data) {Connected()} else { reset -> Skip };
8 System() = User() ||| Box();
```

of password to DBMS and receives the result via the channel `ctr2db` (Line 21). If the password is correct, it notifies Actuator to open the door via the channel `ctr2a` (Line 22); otherwise, it notifies Alarm via the channel `ctr2alm` (Line 23). Actuator will open the door if Controller notifies it, and then it closes the door after a certain time interval (Line 29).

For verifying EGS, we add a User process to model user behavior (Line 7). The system is the interleaving of six components and User. Note that User is just used for verification, and we don't generate code for user behavior (our framework gives designers the flexibility to choose the components to generate software code for). We have verified three assertions: (1) the door never opens if the password is incorrect, (2) the door will eventually open once the correct password is entered, and (3) if the password is incorrect, Alarm will eventually be switched on (Lines 32-35). After the verification, EGS satisfies the three assertions. The detailed model and the generated software code for EGS can be found in [8].

The secure communication box (SCB) is a device for providing secure communication between two clients. SCB is funded by Singapore Defense, and it is confidential. Thus, we only give a prototype here. Listing 1.5 shows the CSP# model for the SCB prototype. After being powered on, SCB is initialized and then waits for a user connection (Line 5). In our framework, we provide the flexibility of implementing channels as interprocess or socket communications. Designers can choose the implementation for each channel. In SCB, the channel `network` is implemented as a socket communication. Once a user connects to SCB, it keeps receiving packets until the user sends a disconnection packet (Line 7). The detailed model and the generated software code can be found in [8].

6 Conclusion and Future Work

In this work, we proposed a framework that can help a system designer to model embedded systems from a high-level modeling language, verify the design of the system, and automatically generate executable software code whose behavior semantics is consistent with the high-level model. With our framework, the development cycle of embedded systems can be significantly reduced. In our future work, we plan to enhance the code generation such that the synthesized software code can be executed on multi-core architectures. We also plan to extend the syntax of the CSP# modeling language as well as its semantics such that system designers can design more featured systems such as real-time systems, probabilistic systems, safety-critical systems, and security protocols.

References

1. Amnell, T., Fersman, L., Mokrushin, E., Petterson, P., Yi, W.: TIMES: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 60–72. Springer, Heidelberg (2004)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)
3. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR*: A Toolset for Specifying and Analyzing Software Requirements. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 526–531. Springer, Heidelberg (1998)
4. Hsiung, P.A., Lin, S.W.: Automatic synthesis and verification of real-time embedded software for mobile and ubiquitous systems. Computer Languages, Systems & Structures 34(4), 153–169 (2008)
5. Hsiung, P.-A., Lin, S.-W., Hung, C.-C., Fu, J.-M., Lin, C.-S., Chiang, C.-C., Chiang, K.-C., Lu, C.-H., Lu, P.-H.: Real-Time Embedded Software Design for Mobile and Ubiquitous Systems. In: Kuo, T.-W., Sha, E., Guo, M., Yang, L.T., Shao, Z. (eds.) EUC 2007. LNCS, vol. 4808, pp. 718–729. Springer, Heidelberg (2007)
6. Hsiung, P.A., Lin, S.W., Tseng, C.H., Lee, T.Y., Fu, J.M., See, W.B.: VERTAF: An application framework for the design and verification of embedded real-time software. IEEE Transactions on Software Engineering 30(10), 656–674 (2004)
7. Knapp, A., Merz, S., Rauh, C.: Model Checking - Timed UML State Machines and Collaborations. In: Damm, W., Olderog, E.-R. (eds.) FTRTFT 2002. LNCS, vol. 2469, pp. 395–414. Springer, Heidelberg (2002)
8. Lin, S.W.: https://sites.google.com/site/shangweilin/pat-codegen
9. Liu, Y., Sun, J., Dong, J.S.: Developing Model Checkers Using PAT. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 371–377. Springer, Heidelberg (2010)
10. Méry, D., Singh, N.K.: Automatic code generation from event-B models. In: SoICT 2011, pp. 179–188 (2011)
11. Niz, D., Rajkumar, R.: Time Weaver: A software-through-models framework for embedded real-time systems. In: LCTES, pp. 133–143 (2003)
12. Peterson, G.L.: Myths about the mutual exclusion problem. Information Processing Letters 10(3), 115–116 (1981)
13. Ramkarthik, S., Zhang, C.: Generating java skeletal code with design contracts from specifications in a subset of object Z. In: ACIS-ICIS 2006, pp. 405–411 (2006)
14. Samek, M.: Practical UML Statecharts in C/C++: Event-Driven Programming for Embedded Systems. Newnes (2008)
15. SCADE, http://www.esterel-technologies.com/products/scade-suite/
16. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: TASE 2009, vol. 962, pp. 127–135 (2009)
17. Thompson, J.M., Heimdahl, M.P.E., Miller, S.P.: Specification-based prototyping for embedded systems. In: SIGSOFT 1999, pp. 163–179 (1999)

Complementary Methodologies for Developing Hybrid Systems with Event-B

Wen Su[1], Jean-Raymond Abrial[2], and Huibiao Zhu[1]

[1] Software Engineering Institute, East China Normal University
{wensu,hbzhu}@sei.ecnu.edu.cn
[2] Marseille, France
jrabrial@neuf.fr

Abstract. This paper contains a further contribution to the handling of hybrid systems as presented in [3]. This time we insist on the usage of multiple methodologies involving not only refinements and proofs as in Event-B and the Rodin Platform, but also Matlab simulation, Animation, and Invariant discovery. We believe that a successful understanding of hybrid systems has to be done in this way by involving several distinct methodologies that are complementary. The paper also presents many examples illustrating the approach.

1 Introduction

By studying the extremely rich literature on hybrid systems, we found that it can be divided, roughly speaking, into two separate groups: those papers dealing with theoretical developments presenting, among others, ways of discovering invariants for hybrid systems whose continuous parts are defined by non-linear differential equations [12] [13] [14] [15], and those papers dealing with practical developments on the verification of hybrid systems whose continuous parts are based on linear differential equations [6] [8] [9] [10][1]. In the present paper we definitely place ourselves in the second group. The reason for this choice is not that we think that the problematics studied and the results presented in the first group are not important, they definitely are, this is rather because we found that the examples presented in the second group are closer to industrial applications than those presented in the first group.

Another reason for our choice of the second group is that the examples presented in this group are usually studied with *one technology* only (mainly model checking). We believe that they could be approached by using *several technologies* that are complementary. In fact, it seems to us that this philosophy is adequate for handling industrial developments where safety is important. We believe that the confidence in the correct development of complex systems is significantly improved by the positive convergence of several independent (but complementary) approaches.

[1] There are obviously *far more* papers in these two groups than the ones cited here.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 230–248, 2012.

Our paper is organized as follows: the next section contains the description of our methodology where we explain how various technologies are applied. The subsequent section contains many examples illustrating our approach. We finally conclude and propose some future research.

2 Methodological Approach

In this section, we develop four techniques for building up a methodology to be used in a systematic fashion for the development of hybrid systems. The first technique (section 2.1) consists in defining a *design pattern* for handling tasks under some timing constraints in Event-B framework [1]. The second technique (in section 2.2) shows how we can take advantage of performing simulations of hybrid systems using *Matlab*. The third technique (in section 2.3) proposes a systematic technique for discovering and handling, so-called, *technical invariants* by means of the prover of the Rodin Platform [2] as well as its animator. The final technique (in section 2.4) shows how ideal hybrid systems can be made more practical by introducing some time constrained *sensors* and *actuators* in between a controller and an environment. It is important to note again that these different approaches are very *complementary*. Each example of section 3 will use several of them.

2.1 Design Patterns for Linear Hybrid Systems

We are interested in presenting a general framework (a formal design pattern as proposed in [4] and [5]) that can be used in the development of hybrid systems where the continuous parts are using some simple *clocks*. This framework will be used under various options in some of the examples proposed in section 3. It can be proved that this approach is similar (although done differently) to the one advocated in continuous Action System [7] [8] [9]: rather than dealing with an absolute time as in [7], we rather handle time with delays.

Such systems are represented by means of a number of *tasks*. We suppose that each task can be either working or idle (boolean variable *work* where $work =$ TRUE means the task is working while $work =$ FALSE means it is idle). A working task can do so for a certain duration (variable *working*). Likewise, an idle task can be sleeping for a certain duration (variable *sleeping*). This can be formalized in Event-B [3] with the following variables and invariants for each task (here $task1$):

inv1: $work1 = \text{TRUE} \Rightarrow sleeping1 = 0$
inv2: $work1 = \text{FALSE} \Rightarrow working1 = 0$

Notice that a task may have no *sleeping* time when it is triggered by another one (example: an actuator task). Likewise, a task my have no *working* time when "execution" is not taken into account. However, such a task might have a sleeping time if it is periodic (example: a sensor task). Besides the initialization, the dynamic behavior of *each task* is formalized by means of two events as follows (here for $task1$):

task1_start
when
$work1 = \text{FALSE}$
$sleeping1 = 0$
then
$work1 := \text{TRUE}$
$working1 := ...$
end

task1_end
when
$work1 = \text{TRUE}$
$working1 = 0$
then
$work1 := \text{FALSE}$
$sleeping1 := ...$
end

The advance of time in the mathematical simulation of such a system is performed by means of a special event called Click that is shown below in the case of two tasks, $task1$ and $task2$. The event Click is enabled when no "normal" event can be enabled any more (this is taken into account in the third guard of the event Click). By doing this, the time makes progress only when all actions to be performed at the "present" time have been "executed". Then, time makes progress by taking the smallest non zero working or sleeping time and removing it from non zero working or sleeping times.

Click
any w **where**
$\{working1, sleeping1, working2, sleeping2\} \setminus \{0\} \neq \varnothing$
$w = \min(\{working1, sleeping1, working2, sleeping2\} \setminus \{0\})$
$\neg((work1 = \text{FALSE} \wedge sleeping1 = 0) \vee (work1 = \text{TRUE} \wedge working1 = 0) \vee$
$(work2 = \text{FALSE} \wedge sleeping2 = 0) \vee (work2 = \text{TRUE} \wedge working2 = 0))$
then
$working1 := \max(\{0, working1 - w\})$
$sleeping1 := \max(\{0, sleeping1 - w\})$
$working2 := \max(\{0, working2 - w\})$
$sleeping2 := \max(\{0, sleeping2 - w\})$
end

For instance if $working1 = 3, sleeping1 = 0, working2 = 0$, and $sleeping2 = 5$, then after Click, $working1 = 0, sleeping1 = 0, working2 = 0$, and $sleeping2 = 2$.

More relationships can be defined between tasks. For instance, there could exist some *priorities* between them. In the case where, say, $task1$ has a priority that is greater than that of $task2$, this is formalized by means of the invariant **inv3** (when $task1$ is working then $task2$ cannot) and the modification of the starting event of $task2$ ($task2$ can only start when $task1$ is sleeping).

inv3: $working1 > 0 \Rightarrow working2 = 0$

task2_start
when
$work2 = \text{FALSE}$
$sleeping2 = 0$
$sleeping1 > 0$
then
$work2 := \text{TRUE}$
$working2 := ...$
end

In case of the introduction of priorities, another relationship may exist between tasks: a working task of lower priority can be *preempted* by a task with greater priority. In the case where, say, $task1$ has a greater priority than that of $task2$ and is able to preempt $task2$, this is formalized by introducing a new variable $remaining2$ together with the following invariants

(*remaining*2 records the time that remains in the execution of *task*2 in case it has been preempted):

inv4: $work2 = \text{FALSE} \Rightarrow remaining2 = 0$
inv5: $working2 = 0 \lor remaining2 = 0$

Then the event task1_start is modified as follows (it possibly preempts *task*2) and a new event task2_restart is introduced in order to restart *task*2 in case it has been previously preempted by *task*1:

task1_start
when
$work1 = \text{FALSE}$
$sleeping1 = 0$
then
$work1 := \text{TRUE}$
$working1 := ...$
$working2 := 0$
$remaining2 := working2$
end

task2_restart
when
$work2 = \text{TRUE}$
$sleeping1 > 0$
$remaining2 > 0$
then
$working2 := remaining2$
$remaining2 := 0$
end

Also the guard $remaining2 = 0$ has to be added to the event task2_stop: a preempted task cannot be stopped. Clearly, the just introduced extensions for taking account of preemption require a corresponding extension of the event Click. In the case where we have more than two events with priorities and multiple preemptions (that is preemption of a preempted task) the previous modifications are a little more elaborate: this can be seen in the example of section 3.4, where all features introduced in this design pattern are shown. Various less complete usage of our design pattern can be seen in the examples of sections 3.1 and 3.3.

2.2 Using Matlab

Matlab Simulink is used to model, simulate and analyze dynamic systems. Moreover, an extension of Simulink, Stateflow, is a special product used for event-driven system. By using Matlab Simulink/Stateflow, we can analyze a model by setting different parameter values.

Our formalization using Matlab follows exactly the last model of each example. This has been done manually for the moment, but we think that it is possible to have a systematic translation from Event-B to Matlab realized in a plug-in.

In the "Water Tank" example we present in section 3.2, we can observe interesting dynamic behaviors under different parameter settings. If the parameters obey the axioms used in the formal development, the system runs normally. In contrast, if these parameters violate the axioms, we can see an abnormal behavior. Also the relationship among such parameters can be analyzed through simulation. More details can be found in section 3.2. We also develop simulation models for other examples presented in this paper to complement the formal development.

2.3 Invariant and Guard Discoveries with the Rodin Platform

When studying the formal development of a system, some of the invariants and event guards are easy to write because they are direct translations of the informal requirements of the system at hand. However, when trying to prove the maintenance of these invariants, one can figure out quite often that it is not possible unless one introduces more invariants that are not so natural than the previous ones. The problem is then to find these new invariants.

An idea, also proposed in [19] is to use the prover for discovering such technical invariants. Here we propose the following: when the prover fails to prove something at a certain point in a proof, one can see where it was blocked (unable to proceed). This is represented by a certain sequent (some hypotheses H together with a goal G: $H \vdash G$) that is impossible to discharge. The simple idea then is to try the following new invariant: $H' \Rightarrow G$, where H' is the conjunction of some of the hypotheses in H.

Introducing this new invariant will discharge the previous proof but requires, of course, to be itself proved. If it is not possible then one can use the same technique, and so on. Usually after a few such iterations, one reachs the "fixpoint" where no new invariants are needed.

The technique we just presented consisted in adding some new invariants. Another technique consists in adding a guard to the event where the invariant maintenance failed. It is a good technique because it does not require more proofs. However, it is not always possible to use as it might change the formalization of the problem and possibly introduce some deadlocks. In the examples of section 3, we shall use both techniques introduced in this section: this is particularly efficient in the example of section 3.3 (mutual exclusion).

An interesting technique to be used while discovering new invariants or new guards is to use the Animators of the Rodin Platform. There are two of them: ProB [18] and AnimB. ProB is more than an animator, it is also a model checker that is quite elaborate. In the formal development of the example presented in this paper, it was sufficient to use AnimB. The idea is to check that the potential addition of an invariant or guard does not introduce some counter-examples. The technique consisting in using the animator as a debugger together with doing formal proofs might seem strange to purists, but we found out that it was very useful as a complementary action.

2.4 Introducing Sensors and Actuators for Refining Ideal Systems

Many examples we found in the literature are dealing with hybrid systems where a controller interacts with an external environment. However, such examples are often ideal in that the controller continuously and directly interacts with its environment. We found that it is quite interesting to put some sensors and actuators between the controller and the environment. Such sensors and actuators might also take some significant time to perform their job. It makes the examples more practical. The price to be paid is that the correct construction can become sometimes quite tricky.

One of the examples below in section 3.2 deals with this question. It appears to us to be more difficult than we expected. In our opinion, more has to be done to find out some systematic ways of handling such situations.

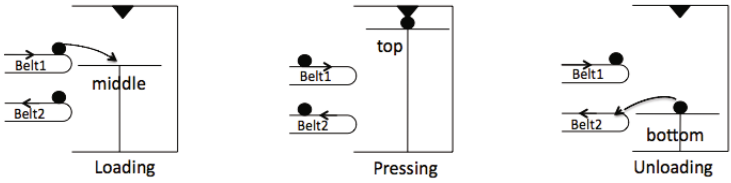
3 Examples

In this section, we present four examples illustrating our usage of the methodology presented in the previous section. The first example in section 3.1 is a trivial application of the the design pattern defined in section 2.1. The second example in section 3.2 illustrates the methodology presented in section 2.4 by introducing sensors in an ideal system. The third and fourth examples in sections 3.3 and 3.4 illustrate more elaborate uses of the technologies presented in section 2.1 and 2.3. Some examples illustrate the usage of Matlab as presented in section 2.2.

3.1 Press

This example is taken from [8] and [17]. It describes a simple model for a mechanical press working in a metal factory. The purpose of this example is to apply the design pattern described in section 2.1 on a reactive system where several devices are working in parallel.

Informal Presentation. The press works as illustrated in the diagrams below. Here is an informal description of this system as quoted from [8]: "The press works as follows. First, its lower part is raised to the middle position. Then an upper conveyor belt feeds a metal blank into the press. When the press is loaded (signaled by *sensor*1 being true), the lower part of the press is raised to the top position and the blank is forged. The press will then move down until the bottom position and the forged blank is placed into a lower conveyor belt. When the press is unloaded (signaled by *sensor*2 being true), the lower part is raised to the middle position for being loaded again."



Defining Some Constants. We start the formal development by defining the three constants *bottom*, *middle*, and *top*. They are numbers with the following obvious axioms: $bottom < middle < top$.

Initial Model. We define a variable *position* with the three possible values (*bottom*, *middle*, and *top*), the two boolean variables for the belt sensors: *sensor*1 and *sensor*2. Finally, we define three "work variables": *bottom_work*, *middle_work*, and *top_work*. When true it means that the corresponding task is active. As a consequence. we have the following invariants:

inv1: $middle\_work = \text{TRUE} \Rightarrow position = middle$
inv2: $top\_work = \text{TRUE} \Rightarrow position = top$
inv3: $bottom\_work = \text{TRUE} \Rightarrow position = bottom$

Next are some events corresponding to task starting or ending:

Loading_start
when
$position = middle$
$sensor1 = \text{TRUE}$
$middle\_work = \text{FALSE}$
then
$sensor1 := \text{FALSE}$
$middle\_work := \text{TRUE}$
end

Loading_end
when
$middle\_work = \text{TRUE}$
then
$position := top$
$middle\_work1 := \text{FALSE}$
end

Belt_1
when
$sensor1 = \text{FALSE}$
then
$sensor1 := \text{TRUE}$
end

During the events Loading_start and Loading_end, the press is moving from middle to top. Similarly, the events Pressing and Unloading (_start and _end) are used to move the press from top to bottom and, from bottom to middle respectively.

First Refinement. In this refinement, we introduce the timing constraints: the three main tasks last for certain times denoted by the variables $middle\_working$, $top\_working$, and $bottom\_working$. Each of them are set to three constants $middle\_t$, top_t, and $bottom\_t$ when started. The two belts might be sleeping (variables $belt1\_sleeping$ and $belt2\_sleeping$) for certain constant times $belt1\_t$ and $belt2\_t$. All these constant times are greater than 0, this are obvious axioms. From the design pattern, we obtain the following invariants:

inv1: $middle\_work = \text{FALSE} \Rightarrow middle\_working = 0$
inv2: $top\_work = \text{FALSE} \Rightarrow top\_working = 0$
inv3: $bottom\_work = \text{FALSE} \Rightarrow bottom\_working = 0$

The events are modified accordingly. We show below, the refined events for the middle task and for the belts.

Loading_start
when
$position = middle$
$sensor1 = \text{TRUE}$
$middle\_work = \text{FALSE}$
then
$sensor1 := \text{FALSE}$
$middle\_work := \text{TRUE}$
$middle\_working := middle\_t$
end

Loading_end
when
$middle\_work = \text{TRUE}$
$middle\_working = 0$
then
$position := top$
$middle\_work := \text{FALSE}$
end

Belt_1
when
$sensor1 = \text{FALSE}$
$belt1\_sleeping = 0$
then
$sensor1 := \text{TRUE}$
$sleeping\_s1 := t\_sen$
end

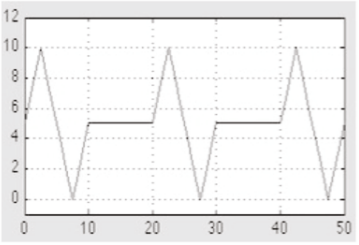
Finally, we write the Click event according to the design pattern of section 2.1.

Click
any w **where**
$\{middle\_working, top\_working, bottom\_working, ...\} \setminus \{0\} \neq \varnothing$
$w = \min(\{middle\_working, top\_working, bottom\_working, ...\} \setminus \{0\})$
"negation of the disjunction of the guards of other events"
then
$mid\_working := \max(\{0, middle\_working - w\})$
$top\_working := \max(\{0, top\_working - w\})$
$bot\_working := \max(\{0, bottom\_working - w\})$
$belt1\_sleeping := \max(\{0, belt1\_sleeping - w\})$
$belt2\_sleeping := \max(\{0, belt2\_sleeping - w\})$
end

Second Refinement. In this refinement (not shown here), the move of the press is done at a certain constant speed v, which gives three axioms, such as $middle\_t = (top - middle) \div v$. Others are similar.

Proof. The overall proof effort of the Rodin platform for this example is the following: 55 proof obligations were generated and proved automatically, except 1 of them proved interactively.

Matlab Simulation. The Matlab simulation is derived from the second refinement, with the following parameters: $bottom = 0, middle = 5, top = 10, middle\_t = 2.5, top\_t = 5, bottom\_t = 2.5, t\_sen = 20$, which strictly obey all the axioms. The right figure is one output of Matlab: it shows how the position (y-axis) evolves with time (x-axis). We see that before loading, the press wait for the sensor of Belt1.

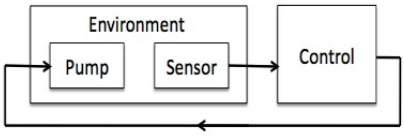


3.2 Water Tank

The example presented in this section is based on the water tank monitoring system presented in [10]. In this example, we illustrate the approaches advocated in sections 2.3 and 2.4.

Informal Presentation. A water tank contains some water that leaks continuously at a certain constant speed $v2$. In order to prevent the water level to become smaller than a certain minimal value tm, a pump can be turned on in order to pour some water in the tank at a certain constant speed, so the resulting speed of leaking and pouring water is $v1$. When the pump is on, it can be turned off in order for the water level to remain under a certain maximal value tM. The purpose of the monitoring is to guarantee that the water level will always be situated within the interval $[tm, tM]$.

In the example described in [10], the water-level is sensed continuously: this is an ideal situation. In the model described here, we are more practical: we suppose that the water level is detected periodically only (every other t_sen seconds) by a sensor. Moreover, we also suppose, as in [10], that the pump takes a certain time, t_act seconds, when activated (or deactivated), in order to be physically effective. The just defined framework is shown in the following figure where one can see the environment made of the pump and the sensor and the controller taking decisions about the status of the pump depending on the information sent to it by the sensor:



Preliminary Study. As already stated, the main invariant of this system is that the water level L in the tank is always situated within the interval $[tm, tM]$. Every other t_sen seconds, the controller is waken up by the sensor with the current level L in the tank. Given the status of the pump (*on* or *off*), the controller has to take a decision: doing nothing, or changing the status of the pump so that the level will remain within the prescribed interval at the next sensor awakening. Suppose the pump is *on*, then in t_sen seconds the water level will be $L + v1 * t\_sen$. If this quantity is smaller than or equal to tM, then we can do nothing. But if this quantity is greater than tM, we have to change the status of the pump. However, as the pump is "lazy" (it take t_act seconds to react), we have to check the next water level against $tM - v1 * t\_act$, not against tM. We have a similar situation when the pump is *off*. In this case, we have to turn it *on* when the quantity $L - v2 * t\_sen$ is smaller than $tm + v2 * t\_act$.

It should be noted that the constants of this system, namely tM, tm, $v1$, $v2$, t_sen, and t_act should obey certain constraints. First of all, they must all be positive quantities with tm smaller than tM. Second, t_act must be smaller than t_sen so that the pump action can make sense. Finally, we have the feeling that the system cannot work if the sampling t_sen is too big. In fact, if the detected level is L and if we have $L + v1 * t\_sen > tM$ but also $L - v2 * t\_sen < tm$, then no decision can be taken by the controller. This happens in the case where $tM - tm < (v1 + v2) * t\_sen$. In other words, in order to strictly avoid this case, the sampling time t_sen has to be such that $t\_sen \leq \frac{tM - tm}{v1 + v2}$.

Defining Some Generic Sets and Constants. We start the formal development by defining the constants previously mentioned. To begin with, we can avoid writing the two axioms $t\_act < t\_sen$ and $tM - tm \geqslant (v1 + v2) * t\_sen$. In this way, we can figure out where these constraints are indispensable in the proofs. We also defined the set P of pump status: $\{on, off\}$.

Formal Model. The formal model is made of three variables L, $PUMP$, and $pump$. The first two variables denote the physical values of the water level and the physical status of the pump. The last variable denotes the status of the pump as ordered by the controller. $PUMP$ and $pump$ are not always identical: this is due to the delay between the command issued by the controller and the effective

physical status of the pump in the environment. A last technical variable, *phase*, with value 1 or 2, denotes the "place" where we are in the simulated model: either in the controller ($phase = 1$) or in the environment ($phase = 2$). Besides the typing of the variable, we have two main invariants that are the followings:

inv1: $phase = 1 \Rightarrow PUMP = pump$
inv2: $L \in tm \,..\, tM$

There are 4 controller events for deciding what should be the next status of the pump. Next are the two events for deciding what to do when the pump is *on*:

decide_1
when
$phase = 1$
$pump = on$
$L + v1 * t\_sen \leq tM - v1 * t\_act$
then
$phase := 2$
end

decide_2
when
$phase = 1$
$pump = on$
$L + v1 * t\_sen > tM - v1 * t\_act$
then
$phase := 2$
$pump := off$
end

We have similar events for deciding what to do when the pump is *off*. In the environment, we have also four events. Here are the events corresponding to the pump receiving the command *on* together with the response made by the environment (the water level L):

env_1
when
$phase = 2$
$pump = on$
$PUMP = on$
then
$phase := 1$
$L := L + v1 * t\_sen$
end

env_2
when
$phase = 2$
$pump = on$
$PUMP = off$
then
$phase := 1$
$PUMP := on$
$L := L - v2 * t\_act + v1 * (t\_sen - t\_act)$
end

The maintenance proof of the first invariant is easy. The invariance proof of the second invariant is more problematic. For instance, the proof of **inv2** by event env_1 fails with the following sequent that is impossible to prove:

$$pump = on, PUMP = on, phase = 2 \;\vdash\; L + v1 * t\_sen \leq tM$$

By applying the simple technique of invariant discovery developed in section 2.3, we add the following invariant **inv3** (and by analogy the invariant **inv4**):

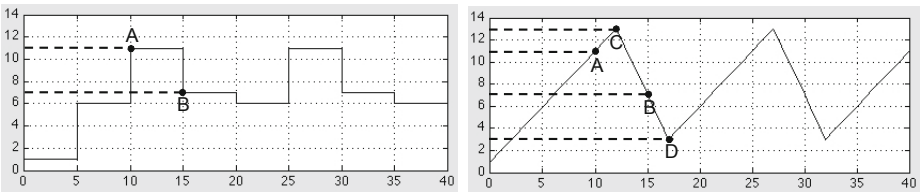
inv3: $pump = on \;\wedge\; PUMP = on \;\wedge\; phase = 2 \;\Rightarrow\; L + v1 * t\_sen \leq tM$
inv4: $pump = off \;\wedge\; PUMP = off \;\wedge\; phase = 2 \;\Rightarrow\; tm \leq L - v2 * t\_sen$

After this, our two failing proofs were discharged but, of course, the proofs of the new invariants failed. We apply similar techniques to prove them and so on. After introducing four new invariants in this way, we reached the fixpoint and all proofs were discharged. In the proofs of these new invariants, we found that the two axioms $t\_act < t\_sen$ and $tM - tm \geqslant (v1 + v2) * t\_sen$ were indispensable.

Refinement. It is possible to refine several times the previous model (not done here) in order to separate the pump and the sensor in the environment and then apply the approach of section 2.1.

Proof. The overall proof effort of the Rodin platform for this example is the following: 77 proof obligations were generated and proved automatically, except 4 of them that were proved interactively.

Matlab Simulation. In the Matlab simulation, we use the following parameters: $tm = 1, tM = 16, t\_sen = 5, t\_act = 2, v1 = 1, v2 = 2$. This parameter are coherent with axioms: $t\_act < t\_sen$ and $tM - tm \geqslant (v1 + v2) \times t\_sen$. We build a model in Stateflow which is an exact translation of the formal development. The time is continuous but the water level is discrete (modified every other t_sen seconds). The corresponding Matlab output can be seen in the first diagram below where the x-axis shows the time and the y-axis shows the value of water level. In order to analyze the continuous behavior of the water level, we refine this model with continuous linear change. The output can be seen in the second diagram below. It is interesting to compare these diagrams. In the second diagram, the water level goes higher (point C) and lower (point D) than in the first one (points A and B), but still between 1 and 16. It is due to the fact that the pump only reacts after t_act second to the command sent by the controller, so the water level continues to augment (with speed 1) or diminish (with speed 2) for t_act (2 seconds) .

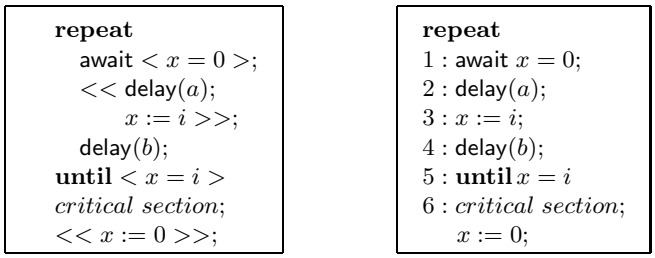


3.3 Mutual Exclusion

This example contains the development of Fischer's mutual exclusion algorithm. We could not find the paper by Fischer where this algorithm would have been introduced (if it ever existed), but it is studied in many occasions in the literature. We take its definition from [11]. The purpose of this example is to illustrate part of the tasking design pattern introduced in section 2.1 and also the guard and invariant discoveries presented in section 2.3.

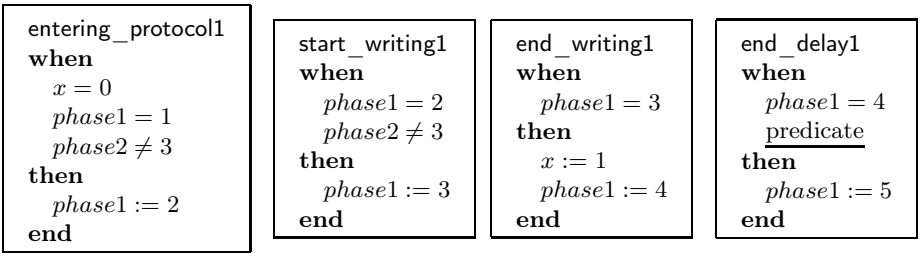
Informal Presentation. We have a number of processes that want to enter *in an exclusive way* a certain piece of code called the "critical section". For that

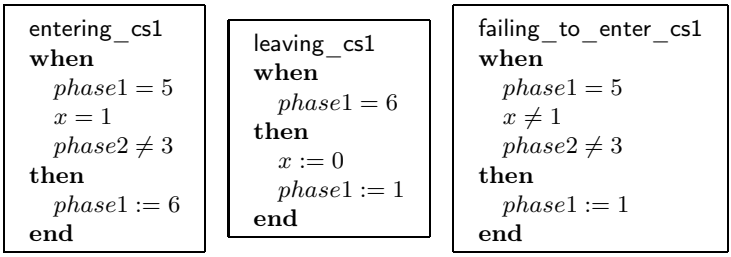
purpose, a variable, x, is *shared* by all these processes. Each process (named according to a distinct positive number i) executes the protocol shown on the left of the figure below. The statements in angle brackets are supposed to be atomic. Double brackets are used around modifications of the variable x: it means that such statements are strictly atomic (only one writer and no reader). Single brackets are used around reading of the variable x: it means that several processes can read x simultaneously (but no writing). The statement $x := i$ takes a certain time a to be executed: this is indicated by prefixing $x := i$ with the statement delay(a). After modifying x, we have to wait for a further delay b. To the right of the figure, the protocol is re-written with a number of addresses prefixing each instruction: these addresses will be used in the formal developments.



We suppose that a is smaller than b and we want to prove that no more than one process can be in the critical section at a time.

Initial Model. In this initial model, we do not introduce the delays a and b. We shall thus prove the mutual exclusion property by introducing "cheating guards" in some events. In order to simplify matters, we suppose that we have two processes only, named 1 and 2. We only define the variable $x \in \{0, 1, 2\}$ and two addresses phases, $phase1$ and $phase2$, corresponding to each processes. Each of them can take a value between 1 and 6. The main property to prove is thus the following: $phase1 \neq 6 \vee phase2 \neq 6$: both processes cannot be simultaneously in the critical section where the phases are equal to 6. Next are the various events for process 1. Similar ones can be defined for process 2. Each event corresponds to one "instruction" of the protocol. As can be seen, we define the guard predicate in the event end_delay1. This guard will be the condition guaranteeing mutual exclusion. It will be made precise at the end of this section.





Rather than defining the mutual exclusion property as an invariant, we use some weaker invariant conditions, namely:

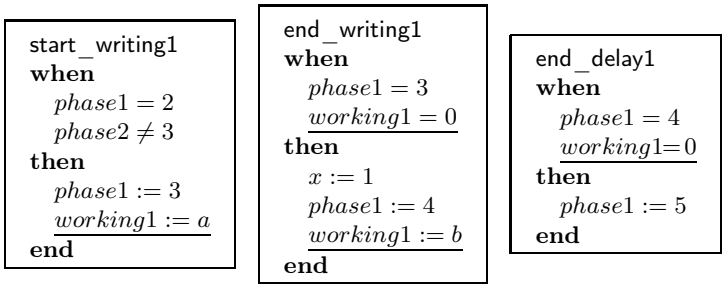
$$\textbf{inv1:}\ \ phase1 = 6 \Rightarrow x = 1 \qquad \textbf{inv2:}\ \ phase2 = 6 \Rightarrow x = 2$$

Then the mutual exclusion property $phase1 \neq 6 \lor phase2 \neq 6$ can be proved as a simple theorem. In order to prove the maintenance of **inv1** and **inv2**, we use again the technique of section 2.3 and discover that the following additional invariants are needed:

$$\textbf{inv3:}\ \ (phase2 = 5 \land x = 2) \lor phase2 = 6 \Rightarrow phase1 \neq 2 \land phase1 \neq 3$$
$$\textbf{inv4:}\ \ (phase1 = 5 \land x = 1) \lor phase1 = 6 \Rightarrow phase2 \neq 2 \land phase2 \neq 3$$

And now, in order to prove the invariants with events end_delay1 and end_delay2 we found (using again the technique of section 2.3) that the guard named "predicate" in the event end_delay1 should be $x = 1 \Rightarrow phase2 \neq 2 \land phase2 \neq 3$, and a similar one in the event end_delay2. In other words, in the case of the event end_delay1, the process 2 should not be entering the writing instruction for x. This is to be taken care when x is 1 only. This is easy to understand: should this guard be missing then process 2 could enter the critical section (with $x = 2$) while process 1 is still in it.

Refinement. Our task in this refinement is clear. The mutual exclusion property has been proved in the abstraction thanks to the introduction of a "cheating" guard ($x = 1 \Rightarrow phase2 \neq 2 \land phase2 \neq 3$) in the event end_delay1 and a similar one in event end_delay2. It remains now for us to remove this cheating guard and simply replace it by the fact that the delay b is over. For this, we follow partly the design pattern defined in section 2.3: we introduce two new variables $working1$ and $working2$ for handling the delays in both processes. Only events start_writing, end_writing and end_delay are then modified as follows:



It remains now for us to introduce the Click event defined in section 2.3 for advancing time in this mathematical simulation:

Click
any w **where**
$\{working1, working2\} \setminus \{0\} \neq \varnothing$
$w = \min(\{working1, working2\} \setminus \{0\})$
"negation of the disjunction of the guards of other events"
then
$working1 := \max(\{0, working1 - w\})$
$working2 := \max(\{0, working2 - w\})$
end

In order to prove the correct refinement of the event end_delay, we have to prove that the concrete guards of end_delay for both processes are *stronger* than those of the corresponding abstractions, that is:

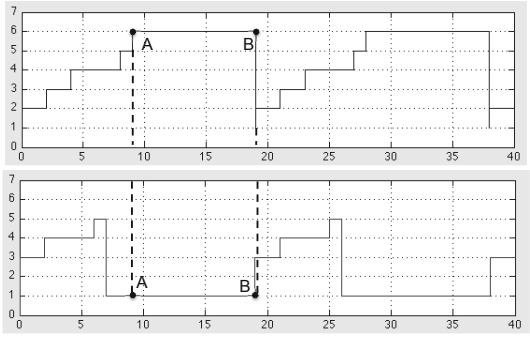
$$phase1 = 4 \ \wedge \ working1 = 0 \ \Rightarrow \ (x = 1 \ \Rightarrow \ phase2 \neq 2 \ \wedge \ phase2 \neq 3)$$
$$phase2 = 4 \ \wedge \ working2 = 0 \ \Rightarrow \ (x = 2 \ \Rightarrow \ phase2 \neq 2 \ \wedge \ phase2 \neq 3)$$

Again, these proofs failed and we have to use the technique of section 2.3 to introduce more technical invariants. This time, it required a little more new invariants than in the abstraction. In fact, seven new invariants were needed before reaching the fixpoint. Since we strengthen the guard, we have also to prove that we have not ended in a possible deadlock. It was easy to prove the deadlock freeness theorem (the disjunction of all guards is true).

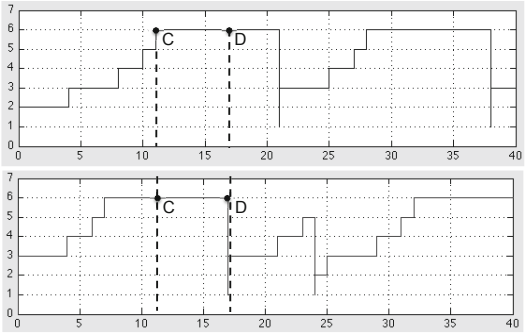
Proof. The overall proof effort of the Rodin platform for this example is the following: 291 proof obligations were generated and all proved automatically.

Matlab Simulation. In the Matlab simulation diagrams below, time is on the x-axis whereas phases are in the y-axis. Each process is in a separate diagram.

In the first Matlab simulation, we have $a = 2$ and $b = 4$, that is $a < b$. We can see the effective mutual exclusion: process 1 is in the critical section between A and B ($phase1 = 6$) while process 2 is not ($phase2 = 1$) during the same time span.



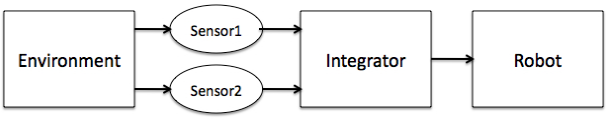
In the second Matlab simulation, we have $a = 4$ and $b = 2$, that is $a \geq b$. Here we can see that mutual exclusion is not achieved. Process 1 and process 2 can be simultaneously in the critical section between C and D ($phase1 = 6$ and $phase2 = 6$).



3.4 Sensors

This example is taken from [16] and [6]. It is described in [6] by means of an algorithm dealing with the tasking features of ADA: tasks (with priority and preemption), rendez-vous, and delays. The purpose of this example is to illustrate the full design pattern introduced in section 2.1.

Informal Presentation. This system is made of two sensors sampling some data from a certain environment. These sensors are connected to an "integrator" that periodically sends an information to a robot. This information is elaborated by the integrator from the data collected by both sensors. This can be illustrated on this figure



For a more precise description of this system, we quote the following paragraph from [6]: "Each sensor constructs a reading and sends it to the integrator. Since the environment is constantly changing, the sensor reading expires if not accepted [by the integrator] within a certain time and a new reading is immediately constructed. When the data has been sent successfully [from a sensor to the integrator], the sensor sleeps for a certain time.The integrator accepts the readings of the two sensors in either order and then computes a command to the robot. To make matters more difficult, there is a proximity requirement: the [two] readings used [by the integrator] to construct a command must have been received within a bounded interval." Notice that the reading of each sensor, the integrator acceptance from a sensor, and the integrator computation of the command for the robot take certain times. Here is a precise specifications of all these timing constraints (unit is 100 micro seconds):

sensor1 expires	40	sensor sleeping	60	integrator accepting	9 .. 10
sensor2 expires	80	sensor 1 reading	5 .. 11	integrator computing	36 .. 56
proximity expires	100	sensor 2 reading	15 .. 20		

As additional requirements, we suppose that the task related to the first sensor has a greater priority than that related to the second sensor. Moreover, the first one can preempt the last one. Finally, the task computing the robot command in the integrator is supposed to have a lower priority than those of the sensors. Likewise, the integrator task can be preempted by sensor tasks.

Defining some Generic Sets and Constants. The formal development starts with the definition of the set D of data acquired by the sensor in the environment and the set S of signals sent to the robot. We also formalize the computation made by the integrator by means of a total function cmp belonging to $S \times S \rightarrow D$.

Initial Model. The initial, most abstract, model has just one variable, $signal$, of type S and a single event Compute defined as indicated in the right. As can be seen, in this inital model we do not take account of the sensors: we suppose that the integrator can access directly to the environment.

Compute
any $d1$ $d2$ **where**
$d1 \in D$
$d2 \in D$
then
$signal := cmp(d1 \mapsto d2)$
end

First Refinement. In the first refinement, we are more precise. We introduce both sensors. Each of them has its own buffer ($read1$ and $read2$, of type D). The connection with the integrator is made through two internal buffers ($acq1$ and $acq2$ of type D). We show below some events dealing with the first sensor reading information from the environment (event sensor1) and being connected with the integrator (event acquire1). Similar events are defined for the second sensor:

sensor1
any d **where**
$d \in D$
then
$read1 := d$
end

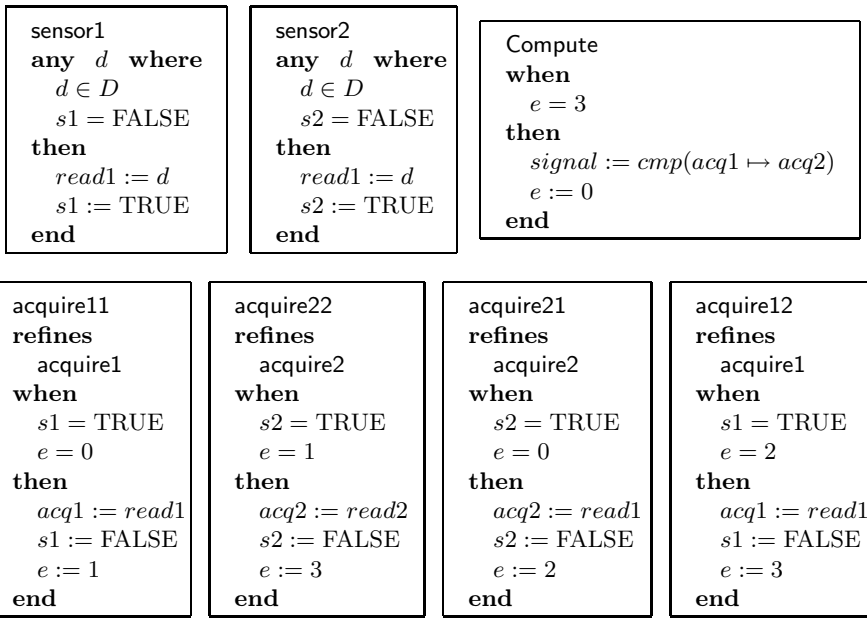
acquire1
begin
$acq1 := read1$
end

Compute
refines
Compute
with
$d1 = acq1$
$d2 = acq2$
then
$signal := cmp(acq1 \mapsto acq2)$
end

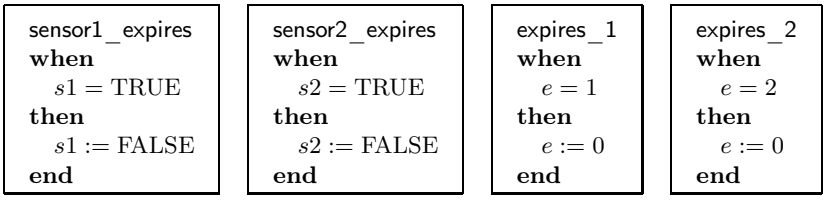
Here we have connected the sensors to the integrator but we have not synchronized yet the sensors with the integrator as required by the informal presentation. This will be done in the next refinement.

Second Refinement. In this refinement, we make more precise the connection of the sensors with the integrator. We now require that the integrator computes

the signal to the robot with data that have been collected by both sensors in the *same round*. We also introduce the possible expiration of sensors and of the integrator. For this, we introduce a boolean variable in each sensor ($s1$ and $s2$). When $s1$ is true, it means that the first sensor has collected a new data that has not yet been sent to the integrator. Finally, we introduce the non-determinacy of the integrator: it can connect the sensor task in any order (sensor1 then sensor2 or the converse).

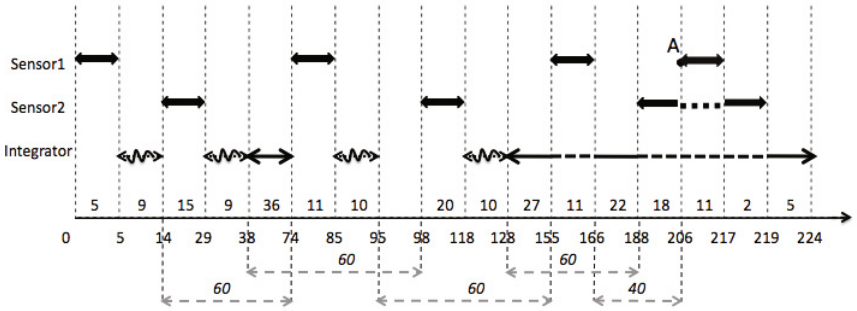


As can be seen the integrator has an "address counter", e, able to take four different values: 0,1,2, and 3. In this refinement, we also introduce possible expirations for the sensors or the integrator. For the moment, such expiration are rather simple and abstract (we do not explain yet why the sensors or the integrator expire):



Third Refinement. In this last refinement, we take account of the various timing and priority constraints described in the informal requirement. For this, we use and follow exactly the development of the design pattern made in section 2.1. Room is lacking here to describe in detail the corresponding development.

Animation. We used the AnimB animator of the Rodin Platform and were able to reproduce exactly the following flow of control described in [6]:



The Integrator task corresponds either to the rendez-vous between the sensors and the integrator or the computation task of the integrator. The 60 delays in the bottom of the diagram denote the sleeping times of the sensor. In point A, we can see that sensor 1 expires (after 40 seconds) because it was not able to connect to the (busy) integrator. After expiration, sensor1 restarts immediately. From times 128 to time 224, we can see (the dashed lines) how the integrator is preempted by sensors 1 and 2 and how sensor 2 is itself preempted by sensor 1.

Proof. The overall proof effort of the Rodin platform for this example is the following: 171 proof obligations were generated and proved automatically, except 8 of them that were proved interactively.

4 Conclusion

In this paper, we presented a methodological framework for developing linear hybrid systems where time is handled by means of various clocks recording task working or sleeping times. Our approach was illustrated with four different examples that were developed using Event-B, the Rodin Platform (examples were fully proved) and also Matlab. The connection between Event-B and Matlab is extremely interesting: we intend to develop a plug-in for the automating translation from the former to the latter. We think that more examples have to be made within this framework so that more design patterns could be developed thus making the engineering of hybrid systems very systematic for industrial applications.

Acknowledgement. This work was partly supported by the Danish National Research Foundation and the National Natural Science Foundation of China (Grant No. 61061130541) for the Danish-Chinese Center for Cyber Physical Systems, also supported by National Basic Research Program of China (No. 2011CB302904), National High Technology Research and Development Program of China (No. 2011AA010101 and No. 2012AA011205), National Natural Science Foundation of China (No. 61021004), and Shanghai Leading Academic Discipline Project (No. B412).

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University press (2010)
2. http://www.event-b.org
3. Abrial, J.-R., Su, W., Zhu, H.: Formalizing Hybrid Systems with Event-B. In: Derrick, J., Fitzgerald, J., Gnesi, S., Khurshid, S., Leuschel, M., Reeves, S., Riccobene, E. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 178–193. Springer, Heidelberg (2012)
4. Abrial, J.-R., Hoang, T.S.: Using Design Patterns in Formal Methods: An Event-B Approach. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 1–2. Springer, Heidelberg (2008)
5. Hoang, T.S., Furst, A., Abrial, J.-R.: Event-B Patterns and Their Tool Support. In: SEFM 2009 (2009)
6. Corbett, J.C.: Modeling and Analysis of Real-Time Ada Tasking Programs. In: IEEE Real-Time Systems Symposium (1994)
7. Back, R.J., Kurki-Suonio, R.: Distributed Cooperation with Action Systems. ACM Transaction on Programming Languages and Systems 10(4), 513–554 (1988)
8. Back, R.-J., Petre, L., Porres, I.: Generalizing Action Systems to Hybrid Systems. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 202–213. Springer, Heidelberg (2000)
9. Back, R.J., Cerschi Seceleanu, C., Westerholm, J.: Symbolic Simulation of Hybrid Systems. In: APSEC 2002 (2002)
10. Alur, R., et al.: The Algorithmic Analysis of Hybrid Systems. Theoretical Computer Science 138, 3–34 (1995)
11. Lamport, L.: A fast mutual exclusion Algorithm. ACM Transactions on Computer Systems (1987)
12. Lin, W., Wu, M., Yang, Z., Zeng, Z.: Exact Safety Verification of Hybrid Systems Using Sums-Of-Squares Representation. CoRR 2011 (2011)
13. Ratschan, S., She, Z.: Safety Verification of Hybrid Systems by Constraint Propagation Based Abstraction Refinement. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 573–589. Springer, Heidelberg (2005)
14. Sankaranarayanan, S., Sipma, H., Manna, Z.: Constructing Invariants for Hybrid Systems. Formal Methods in System Design, SSM04b 2008 (2008)
15. Liu, J., Zhan, N., Zhao, H.: Computing semi-algebraic invariants for polynomial dynamical systems. In: EMSOFT 2011 (2011)
16. Gerber, R., Lee, I.: A layered approach to automating verification of real-time systems. IEEE Transaction on Software Engineering (1992)
17. Lewerentz, C., Lindner, T. (eds.): Formal Development of Reactive Systems. LNCS, vol. 891. Springer, Heidelberg (1995)
18. Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. In: STTT 2008 (2008)
19. Ogata, K., Futatsugi, K.: Some Tips on Writing Proof Scores in the OTS/CafeOBJ Method. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 596–615. Springer, Heidelberg (2006)

A Temporal Logic with Mean-Payoff Constraints

Takashi Tomita[1], Shin Hiura[2], Shigeki Hagihara[1], and Naoki Yonezaki[1]

[1] Tokyo Institute of Technology, Tokyo, Japan
{tomita,hagihara,yonezaki}@fmx.cs.titech.ac.jp
[2] NS Solutions Corporation, Tokyo, Japan

Abstract. In the quantitative verification and synthesis of reactive systems, the states or transitions of a system are associated with payoffs, and a quantitative property of a behavior of the system is often characterized by the mean payoff for the behavior. This paper proposes an extension of LTL that describes mean-payoff constraints. For each step of a behavior of a system, the payment depends on a system transition and a temporal property of the behavior. A mean-payoff constraint is a threshold condition for the limit supremum or limit infimum of the mean payoffs of a behavior. This extension allows us to describe specifications reflecting qualitative and quantitative requirements on long-run average of costs and the frequencies of satisfaction of temporal properties. Moreover, we develop an algorithm for the emptiness problems of multi-dimensional payoff automata with Büchi acceptance conditions and multi-threshold mean-payoff acceptance conditions. The emptiness problems are decided by solving linear constraint satisfaction problems, and the decision problems of our logic are reduced to the emptiness problems. Consequently, we obtain exponential-time algorithms for the model- and satisfiability-checking of the logic. Some optimization problems of the logic can also be reduced to linear programming problems.

Keywords: LTL, automata, mean payoff, formal verification, decision problems, specification optimization, linear programming.

1 Introduction

Research on the formal verification and synthesis of reactive systems has focused on the qualitative properties of behaviors (e.g., "undesirable properties never hold" and "some properties hold infinitely often"). Linear Temporal Logic [19] (LTL), which is a subset of the class of ω-regular languages (i.e., languages recognized by finite-state automata such as Büchi automata and Rabin automata), is widely used to describe such properties. For LTL specifications, several model- and realizability- [18] checkers (e.g., SPIN [21] and Acacia+ [1], respectively) have been provided.

Alternatively, as an approach for describing quantitative properties, quantitative languages [15,17,12,2,11] have recently been proposed. A quantitative language is a function that gives a value in a certain ordered range to each word.[1]

[1] It is a Boolean language if the range is Boolean.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 249–265, 2012.

In the models of these languages, a payoff (or weight/cost/reward) is associated with transitions or states. Some quantitative attributes of a system behavior (e.g., the long-run average cost and the frequency of being in unexpected states) can be characterized as certain values pertaining to the mean payoff of the behavior. In quantitative synthesis [14,7,13,10], a program or strategy is optimized for such a value in the ordered range.

Alur et al. proposed a multi-threshold mean-payoff language [2], as a tractable Boolean language for describing quantitative aspects of behaviors. This language is recognized by a payoff automaton with a *multi-threshold mean-payoff acceptance condition*. A payoff is a real vector associated with a transition of the automaton. It accepts a word over a run satisfying the mean-payoff acceptance condition, given by a Boolean combination of *threshold conditions* (i.e., inequalities relating a constant threshold and the maximum or minimum value of the interval of a certain coordinate projection of accumulation points of mean payoffs of the run). The closure property under Boolean operations and the decidability of the emptiness problem for the language have been proved in [2]. However, the languages are incompatible with ω-regular languages, and cannot capture qualitative fairness, such as "a certain property holds infinitely often". Boker et al. proposed $\mathrm{LTL}^{\mathrm{lim}}$ [8], which is an extension of LTL with *path-accumulation assertions* (mean-payoff assertions). In a manner analogous to the multi-threshold mean-payoff languages, a path-accumulation assertion $\mathsf{LimSupAvg}(v) \geq c$ (resp. $\mathsf{LimInfAvg}(v) \geq c$) is a threshold condition; i.e., an inequality relating a constant c and the limit supremum (resp., limit infimum) of mean payoffs for v, where v is a numeric variable whose value depends on the state of a system. They also presented a model-checking algorithm for $\mathrm{LTL}^{\mathrm{lim}}$ against quantitative Kripke structures (in other words, multi-dimensional weighted transition systems). In this algorithm, model-checking is modified to the emptiness problem in [2], considering the Büchi condition reflecting an LTL portion of a specification. Consequently, $\mathrm{LTL}^{\mathrm{lim}}$ allows us to check whether a system satisfies a specification which reflects both qualitative and quantitative requirements. However, mean-payoff assertions are almost meaningless for satisfiability-checking, because either a combination of assertions is inconsistent according to algebraic rules or there exists a trivial variable assignment for which the assertions are true.

This paper is aimed to develop a temporal logic that can describe both qualitative and quantitative properties, and can be used as a verifiable specification language for realizability-checking and synthesis. We propose $\mathrm{LTL}^{\mathrm{mp}}$, which is an extension of $\mathrm{LTL}^{\mathrm{lim}}$ with a payment for satisfying temporal properties. In this logic, for each step of a behavior of a system, the payoff depends not only on a system transition but also on a temporal property of the behavior. Concretely, a payment t consists of free variables $v_1, \ldots, v_n$ (for associating with the transitions of a system), *characteristic variables* $\mathbf{1}_{\varphi_1}, \ldots, \mathbf{1}_{\varphi_m}$ for formulae $\varphi_1, \ldots, \varphi_m$ in the logic (i.e., each $\mathbf{1}_{\varphi_i} = 1$ if φ_i holds at the time, and otherwise $\mathbf{1}_{\varphi_i} = 0$), and algebraic operations. The mean-payoff formula has a form $\overline{\mathsf{MP}}(t) \sim c$ ($\equiv \mathsf{LimSupAvg}(t) \sim c$) or $\underline{\mathsf{MP}}(t) \sim c$ ($\equiv \mathsf{LimInfAvg}(t) \sim c$) for a

payment t and $\sim \in \{<, >, \leq, \geq\}$. LTL$^{\mathrm{mp}}$ can represent the quantitative properties; e.g., "the frequency of satisfying φ is bounded below by 0.1" is represented by $\underline{\mathsf{MP}}(\mathbf{1}_{\varphi}) > 0.1$, and "the long-run average cost is bounded above by 3" is expressed by $\overline{\mathsf{MP}}(6 \cdot \mathbf{1}_{\neg on \wedge \mathbf{X} on} + 4 \cdot \mathbf{1}_{on} + 5 \cdot \mathbf{1}_{on \wedge \mathbf{X} \neg on}) < 3$ if the operating cost is 4 and additional costs for booting and shutdown are 6 and 5, respectively. In addition, we can check the satisfiability of specifications with such meaningful mean-payoff constraints that have no free variable.

We reduce the decision problems of this logic to the emptiness problems of payoff automata Büchi conditions and with multi-threshold mean-payoff conditions. This type of emptiness problem can be also decided by a part of the algorithm in [8]. However, the complexity of that algorithm is roughly estimated to be exponential with respect to the size of the state space of the automaton. Therefore, we develop an algorithm for the emptiness problems of the automata, by reducing these problems to *linear constraint satisfaction problems* (LCSPs). In terms of LCSPs, the difference between the two algorithms is explained as follows: in their algorithm, the solution region is computed explicitly for finding some solutions, whereas our algorithm captures the region implicitly via linear constraints, and then finds the solutions. With this reduction, the emptiness problem of an automaton is decidable in polynomial time for the state space of the automaton. As a result, we obtain exponential-time algorithms for the model- and satisfiability-checking of the logic.

An additional advantage of this reduction is that some optimization problems concerning LTL$^{\mathrm{mp}}$ specifications can be solved via *linear programming* (LP) techniques, which are widely used and well-studied optimization methods. For example, maximization/minimization problems for the limit supremum $\overline{\mathsf{MP}}(t)$ (or limit infimum $\underline{\mathsf{MP}}(t)$) of the mean payoff for a payment t, which is subject to a specification described in LTL$^{\mathrm{mp}}$, are reduced to LP problems. Consequently, we can analyze performance limitations under specifications. We conjecture that this specification optimization method can be applied to realizability-checking as well as optimal synthesis for specifications described in the logic.

Related Work. [12,2,11] introduced quantitative languages focusing on mean-payoff properties. The multi-threshold mean-payoff language [2] and LTL$^{\mathrm{lim}}$ [8] have been proposed as Boolean languages for describing mean-payoff properties. A multi-threshold mean-payoff language can represent threshold mean-payoff properties and some qualitative properties. LTL$^{\mathrm{lim}}$ is an LTL extension with threshold mean-payoff assertions for payoffs associated with transitions of a model. LTL$^{\mathrm{lim}}$ can be used as a specification language for model-checking. However, the mean-payoff assertions are almost meaningless for satisfiability-checking. This paper introduces LTL$^{\mathrm{mp}}$, which is an extension of LTL$^{\mathrm{lim}}$ with payments for satisfying temporal properties. LTL$^{\mathrm{mp}}$ can represent quantitative properties which are meaningful for satisfiability-checking.

In existing methods [14,7,13,10] for the quantitative synthesis, a program (resp., strategy) is synthesized from a partial program or deterministic automaton (resp., Markov decision process or game). A probabilistic environment is

often assumed [14,13,10], and a synthesized program (or strategy) is optimal in the average case. The notion of probability is also introduced in quantitative verification. Probabilistic temporal logics [16,4,5] (and their reward extensions [6,3]) are often used as specification languages, and some probabilistic model-checking tools (e.g., PRISM [20]) have been provided. However, the decidability of their satisfiability problems is an open question.[2] This paper provides an optimization method of $\mathrm{LTL}^{\mathrm{mp}}$ specifications, and we conjecture that our approach to the specification optimization can be applied to optimal synthesis for temporal logic specifications in which quantitative properties are described.

Previously, we introduced a probabilistic temporal logic, with a frequency operator that can describe quantitative linear-time properties pertaining only to conditional frequencies of satisfaction of temporal properties [22]. By contrast, $\mathrm{LTL}^{\mathrm{mp}}$ is a non-probabilistic linear-time logic with mean-payoff formulae. A payment for a mean-payoff formula can be flexibly described. Therefore, the mean-payoff formulae can be used to represent linear-time properties pertaining not only to conditional frequencies, but also to other types of frequencies, such as long-run average costs. (However, the semantics of the mean-payoff formulae are incompatible with those of the frequency operator.)

Organization of the Paper. In Section 2, we introduce the syntax and semantics of $\mathrm{LTL}^{\mathrm{mp}}$, which is an extension of $\mathrm{LTL}^{\mathrm{lim}}$ with payments for satisfying temporal properties. In Section 3, we provide definitions and related notions of payoff automata that accept words over runs satisfying both Büchi conditions and multi-threshold mean-payoff conditions. In addition, we develop an algorithm for the emptiness problems of the automata, in which the problems are reduced to LCSPs. In Section 4, we show how to construct an automaton that recognizes a given $\mathrm{LTL}^{\mathrm{mp}}$ formula, and how to reduce the decision problems of $\mathrm{LTL}^{\mathrm{mp}}$ to the emptiness problems of the automata. We also show that some optimization problems of $\mathrm{LTL}^{\mathrm{mp}}$ specifications can be solved by LP methods. Our conclusions are stated in Section 5.

2 LTL with Mean-Payoff Constraints

In this section, we introduce the syntax and semantics of $\mathrm{LTL}^{\mathrm{mp}}$, which is an extension of $\mathrm{LTL}^{\mathrm{lim}}$ [8] with payments for satisfying temporal properties. In $\mathrm{LTL}^{\mathrm{lim}}$, an assertion has the form either $\mathsf{LimSupAvg}(v) \sim c$ or $\mathsf{LimInfAvg}(v) \sim c$ for a variable v associated with transitions of the system. In comparison, in $\mathrm{LTL}^{\mathrm{mp}}$, a payment for each step of a behavior of a system depends not only on a transition of the system, but also on a temporal property of the behavior. An assertion in $\mathrm{LTL}^{\mathrm{mp}}$ has the form either $\overline{\mathsf{MP}}(t) \sim c$ ($\equiv \mathsf{LimSupAvg}(t) \sim c$) or $\underline{\mathsf{MP}}(t) \sim c$ ($\equiv \mathsf{LimInfAvg}(t) \sim c$), for a payment t consisting of free variables for associating with transitions of the system, characteristic variables associated with temporal properties of the behavior, and algebraic operations.

[2] For the qualitative fragment of Probabilistic CTL [16], the satisfiability problem is decidable [9].

First, we define the syntax of $\mathrm{LTL}^{\mathrm{mp}}$. In the following discussion, we fix the set AP of atomic propositions.

Definition 1 (Syntax). *LTL^{mp} over a set V of free variables is defined inductively as follows:*

$$\begin{aligned} \varphi &::= p \mid \overline{\mathsf{MP}}(t) \sim c \mid \underline{\mathsf{MP}}(t) \sim c \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathbf{X}\varphi \mid \varphi \mathbf{U} \varphi \\ t &::= v \mid \mathbf{1}_{\varphi} \mid t + t \mid -t \mid t \cdot t \mid c \cdot t \end{aligned}$$

where $p \in AP$, $v \in V$, $\sim \in \{<, >, \leq, \geq\}$ and $c \in \mathbb{R}$.

The operators $\mathbf{X}$ and $\mathbf{U}$ are standard temporal operators representing "next" and "until", respectively. Intuitively, $\mathbf{X}\varphi$ means that "φ holds in the *next* step", and $\varphi_1 \mathbf{U} \varphi_2$ means that "φ_2 holds eventually and φ_1 holds *until* then". A payment t consists of free variables $v_1, \ldots, v_n \in V$, characteristic variables $\mathbf{1}_{\varphi_1}, \ldots, \mathbf{1}_{\varphi_m}$ for formulae $\varphi_1, \ldots, \varphi_m$, and algebraic operators ($+$, $-$ and $\cdot$). The major difference between $\mathrm{LTL}^{\mathrm{mp}}$ and $\mathrm{LTL}^{\mathrm{lim}}$ is the existence of characteristic variables. A characteristic variable $\mathbf{1}_{\varphi}$ for a formula φ represents a payment for satisfying the property φ; i.e., $\mathbf{1}_{\varphi} = 1$ if φ holds at the given time, and otherwise $\mathbf{1}_{\varphi} = 0$. The satisfaction of φ at a given time depends on a temporal property of the present and future. In this sense, a characteristic variable is bounded. A free variable v is used for associating with transitions of a system, and an $\mathrm{LTL}^{\mathrm{mp}}$ formula is a *sentence* if it has no free variable. Intuitively, $\overline{\mathsf{MP}}(t)$ and $\underline{\mathsf{MP}}(t)$ give the limit supremum and limit infimum, respectively, of the mean payoff for t. The formulae $\overline{\mathsf{MP}}(t) \sim c$ and $\underline{\mathsf{MP}}(t) \sim c$ are called *mean-payoff formulae*, and are *simple* if t is constructed without characteristic variables for mean-payoff formulae.

We allow common abbreviations of normal logical symbols ($\mathtt{tt} \equiv \varphi \vee \neg\varphi$ and $\mathtt{ff} \equiv \neg\mathtt{tt}$), and connectives ($\varphi_1 \wedge \varphi_2 \equiv \neg(\neg\varphi_1 \vee \neg\varphi_2)$, $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$ and $\varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \rightarrow \varphi_2) \wedge (\varphi_2 \rightarrow \varphi_1)$), and standard temporal operators ($\Diamond\varphi \equiv \mathtt{tt}\mathbf{U}\varphi$ and $\Box\varphi \equiv \neg\Diamond\neg\varphi$). Intuitively, $\Diamond\varphi$ (resp., $\Box\varphi$) means that "φ eventually (resp., always) holds". We also use c instead of $c \cdot \mathbf{1}_{\mathtt{tt}}$, for short.

$\mathrm{LTL}^{\mathrm{mp}}$ can represent a combination of qualitative properties described in classical LTL and quantitative properties given by mean-payoff formulae. We present some simple examples of quantitative properties.

Example 1 (Conditional frequency). A mean-payoff formula for the payment $t = (c_1 \cdot \mathbf{1}_{\varphi_1} - c_2 \cdot \mathbf{1}_{\neg\varphi_1}) \cdot \mathbf{1}_{\varphi_2}$ can represent a property pertaining to the conditional frequency of satisfaction of φ_1 under the condition φ_2, where $c_1, c_2 > 0$. Our previous work [22] focused on the conditional frequencies of satisfying temporal properties and introduced a new binary temporal operator to describe only this type of property. For $\varphi_1 = \mathbf{X}\mathit{response}$ and $\varphi_2 = \mathit{request}$, the formula $\underline{\mathsf{MP}}(t) > 0$ means that "the occurrence frequency of *requests* is not negligible (i.e., $\underline{\mathsf{MP}}(\mathbf{1}_{\mathit{request}}) > 0$) and the limit infimum of the conditional frequency of *responding* to *requests* in the next step is greater than $\frac{c_2}{c_1+c_2}$".

Example 2 (Long-run average costs). Usually, a cost is associated with an event, which has a corresponding proposition. A property of the long-run average of

event-based costs is expressed as a mean-payoff formula for a payment $t = \sum c_i \cdot \mathbf{1}_{p_i}$, where p_i is a proposition representing the occurrence of an event e_i and c_i is the cost for the event e_i. For example, $\overline{\mathsf{MP}}(t) \leq 5$ means that "the long-run average of costs obeying t is bounded above by 5". In addition, switching costs for p_i are described by the characteristic variables $\mathbf{1}_{p_i \wedge \mathbf{X}\neg p_i}$ and $\mathbf{1}_{\neg p_i \wedge \mathbf{X} p_i}$.

Next we define the semantics of $\mathrm{LTL}^{\mathrm{mp}}$.

Definition 2 (Semantics). *For an infinite word $\sigma = a_0 a_1 \cdots \in (2^{AP})^\omega$, an LTL^{mp} formula φ over a set V of free variables, and an assignment $\alpha : V \to \mathbb{R}^\omega$, the satisfaction relation $\models$ is defined inductively as follows:*

$$
\begin{aligned}
\sigma, \alpha, i &\models p \Leftrightarrow p \in a_i, \\
\sigma, \alpha, i &\models \neg\varphi \Leftrightarrow \sigma, \alpha, i \not\models \varphi, \\
\sigma, \alpha, i &\models \varphi_1 \vee \varphi_2 \Leftrightarrow \sigma, \alpha, i \models \varphi_1 \text{ or } \sigma, \alpha, i \models \varphi_2, \\
\sigma, \alpha, i &\models \mathbf{X}\varphi \Leftrightarrow \sigma, \alpha, i+1 \models \varphi, \\
\sigma, \alpha, i &\models \varphi_1 \mathbf{U} \varphi_2 \Leftrightarrow \exists j \geq i. (\sigma, \alpha, j \models \varphi_2 \text{ and } \forall k \in [i, j). \sigma, \alpha, k \models \varphi_1), \\
\sigma, \alpha, i &\models \overline{\mathsf{MP}}(t) \sim c \Leftrightarrow \limsup_{n\to\infty} \tfrac{1}{n+1} \cdot \textstyle\sum_{m=0}^{n} [\![t]\!]_\sigma^\alpha(i+m) \sim c, \\
\sigma, \alpha, i &\models \underline{\mathsf{MP}}(t) \sim c \Leftrightarrow \liminf_{n\to\infty} \tfrac{1}{n+1} \cdot \textstyle\sum_{m=0}^{n} [\![t]\!]_\sigma^\alpha(i+m) \sim c,
\end{aligned}
$$

$$
[\![v]\!]_\sigma^\alpha(i) = \alpha(v)[i] \text{ for } v \in V, \qquad [\![\mathbf{1}_\varphi]\!]_\sigma^\alpha(i) = \begin{cases} 1 & \text{if } \sigma, \alpha, i \models \varphi, \\ 0 & \text{otherwise,} \end{cases}
$$

$$
\begin{aligned}
[\![t_1 + t_2]\!]_\sigma^\alpha(i) &= [\![t_1]\!]_\sigma^\alpha(i) + [\![t_2]\!]_\sigma^\alpha(i), & [\![-t]\!]_\sigma^\alpha(i) &= -[\![t]\!]_\sigma^\alpha(i), \\
[\![t_1 \cdot t_2]\!]_\sigma^\alpha(i) &= [\![t_1]\!]_\sigma^\alpha(i) \cdot [\![t_2]\!]_\sigma^\alpha(i), & [\![c \cdot t]\!]_\sigma^\alpha(i) &= c \cdot [\![t]\!]_\sigma^\alpha(i),
\end{aligned}
$$

where, for an infinite sequence $x = x_0 x_1 \cdots \in \mathbb{R}^\omega$ of real numbers, we denote by $x[i]$ the i-th element of x.

We omit i and/or α from $\sigma, \alpha, i \models \varphi$ if $i = 0$ and/or $V = \emptyset$.

The semantics of mean-payoff formulae are expressed by the limit supremum or limit infimum, and hence, for any word and assignment, the truth-value of a mean-payoff formula is either always true or always false. In a manner analogous to $\mathrm{LTL}^{\mathrm{lim}}$, a formula φ with a mean-payoff subformula ψ is equivalent to a formula $(\varphi[\psi/\mathtt{tt}] \wedge \psi) \vee (\varphi[\psi/\mathtt{ff}] \wedge \neg\psi)$. Furthermore, any payment over $\mathrm{LTL}^{\mathrm{mp}}$ can be represented in the form $\sum(c_i \cdot \mathbf{1}_{\varphi_i} \cdot \prod v_{ij})$. Therefore, we can restrict the syntax of $\mathrm{LTL}^{\mathrm{mp}}$, without loss of generality, to the form $\bigvee(\varphi_i \wedge \bigwedge \psi_{ij})$, where each φ_i is a classical LTL formula (not necessarily conjunctive), each ψ_{ij} is a simple mean-payoff formula, and each payment for ψ_{ij} is of the form $\sum(c_{ijk} \cdot \mathbf{1}_{\varphi_{ijk}} \cdot \prod v_{ijkl})$. We call such a form a *mean-payoff normal form* (MPNF). An $\mathrm{LTL}^{\mathrm{mp}}$ formula φ with n mean-payoff formulae can be transformed, at worst, into an equivalent MPNF formula with 2^n disjuncts, where each distinct has one LTL formula φ_i ($|\varphi_i| \leq |\varphi|$) and n simple mean-payoff formulae.

3 Multi-threshold Mean-Payoff Büchi Automata

In [8], model-checking for an $\mathrm{LTL}^{\mathrm{lim}}$ formula is modified to the emptiness problem of a multi-dimensional payoff automaton with a multi-threshold mean-payoff

condition [2], considering the Büchi condition reflecting the LTL portion of the formula. In this paper, we define payoff automata with both Büchi conditions and multi-threshold mean-payoff conditions. Such automata are called *multi-threshold mean-payoff Büchi automata* (MTMPBAs). In Subsection 3.1, we introduce definitions and concepts related to the automata. The decision problems of $\mathrm{LTL}^{\mathrm{mp}}$ can be reduced to the emptiness problems of the automata, and it can be solved via the part of the algorithm in [8], but with a high complexity. In Subsection 3.2, we develop an algorithm for solving the emptiness problem, using a different approach with lower complexity than that of [8].

3.1 Definitions

In this subsection, we introduce the definitions of the payoff systems and MTMPBAs, together with some concepts related to them.

A payoff system is a multi-dimensional weighted transition system. It is used as a model in quantitative verification.

Definition 3. *A d-dimensional payoff system PS is a tuple $\langle Q, \Sigma, \Delta, q_0, \mathbf{w}\rangle$, where Q is a finite set of states, Σ is a finite alphabet, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, $q_0 \in Q$ is an initial state, and $\mathbf{w} : \Delta \to \mathbb{R}^d$ is a weight function that maps each transition to a d-dimensional real vector. We denote by $\mathbf{w}[i]$ the i-th coordinate function of $\mathbf{w}$; i.e., $\mathbf{w}(\delta) = \langle \mathbf{w}[1](\delta), \ldots, \mathbf{w}[d](\delta)\rangle$.*

For a transition $\delta = \langle q, a, q'\rangle \in \Delta$, we denote by $pre(\delta)$ the pre-state q, $post(\delta)$ the post-state q', and $letter(\delta)$ the letter a. A finite *run* r on Δ is a finite sequence $\delta_0 \cdots \delta_n \in \Delta^*$ of transitions such that $post(\delta_i) = pre(\delta_{i+1})$ for $0 \leq i < n$. A finite *word* σ $(= word(r))$ over a finite run $r = \delta_0 \cdots \delta_n$ is a finite sequence $letter(\delta_0) \cdots letter(\delta_n) \in \Sigma^*$ of letters. A (d-dimensional) finite *trace* τ is a finite sequence of (d-dimensional) real vectors. We denote by $payoff_{\mathbf{w}}(r)$ the trace $\mathbf{w}(\delta_0) \cdots \mathbf{w}(\delta_n)$ of payoffs, and by $mp_{\mathbf{w}}(r)$ the trace $\mathbf{w}(\delta_0) \cdots (\frac{1}{n+1} \sum_{i=0}^{n} \mathbf{w}(\delta_i))$ of mean payoffs, over a finite run $r = \delta_0 \cdots \delta_n$ for a d-dimensional weighted function $\mathbf{w}$. Infinite runs, words and traces are defined in a manner analogous to the finite case. We denote by $run(\Delta)$ the set of finite or infinite runs on Δ, and by $run(PS)$ the set of infinite runs starting from the initial state q_0 and belonging to $run(\Delta)$. A finite run $r = \delta_0 \cdots \delta_n \in run(\Delta)$ is *cyclic* if $pre(\delta_0) = post(\delta_n)$. A state q is *reachable* from q' on Δ if $q = q'$ or there exists a finite run $\delta_0 \cdots \delta_n \in run(\Delta)$ such that $pre(\delta_0) = post(\delta_n)$. A subgraph $\langle Q', \Delta'\rangle$ is a *strongly connected component* (SCC) on PS if $\Delta' \subseteq \Delta \cap Q' \times \Sigma \times Q'$, and for any two states in Q', one is reachable from the other on Δ'.

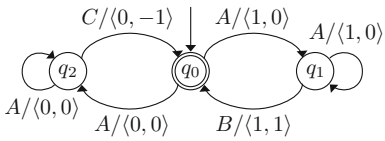


Fig. 1. Example 3

Example 3. Consider the payoff system $PS = \langle Q, 2^{\{p_0,p_1\}}, \Delta, q_0, w\rangle$, where $Q = \{q_0, q_1, q_2\}$, $A = \emptyset$, $B = \{p_0\}$, $C = \{p_1\}$, $\Delta = \{\delta_1, \ldots, \delta_6\}$, $\delta_1 = \langle q_0, A, q_1\rangle$, $\delta_2 = \langle q_1, A, q_1\rangle$, $\delta_3 = \langle q_1, B, q_0\rangle$, $\delta_4 = \langle q_0, A, q_2\rangle$, $\delta_5 = \langle q_2, A, q_2\rangle$, $\delta_6 = \langle q_2, C, q_0\rangle$, $\mathbf{w}[1](\delta) = 1$ if $\delta \in \{\delta_1, \delta_2, \delta_3\}$ and otherwise $\mathbf{w}[1](\delta) = 0$,

and $\mathbf{w}[2](\delta_3) = 1$, $\mathbf{w}[2](\delta_6) = -1$ and $\mathbf{w}[2](\delta) = 0$ if $\delta \in \{\delta_1, \delta_2, \delta_4, \delta_5\}$ (Fig. 1). Consider runs $r_1 = (\delta_1\delta_3)^1\delta_4\delta_6(\delta_1\delta_3)^2\delta_4\delta_6(\delta_1\delta_3)^3\delta_4\delta_6\ldots$ and $r_2 = (\delta_1\delta_3)(\delta_4\delta_5^{2^2-2}\delta_6)(\delta_1\delta_2^{2^3-2}\delta_3)(\delta_4\delta_5^{2^4-2}\delta_6)\ldots$. Then, the trace of payoffs over r_1 is $(\langle 1,0\rangle\langle 1,1\rangle)^1\langle 0,0\rangle\langle 0,\text{-}1\rangle(\langle 1,0\rangle\langle 1,1\rangle)^2\langle 0,0\rangle\langle 0,\text{-}1\rangle(\langle 1,0\rangle\langle 1,1\rangle)^3\langle 0,0\rangle\langle 0,\text{-}1\rangle\ldots$, and the trace of mean payoffs over r_1 converges to the point $\langle 1, 1/2\rangle$. The trace of payoffs over r_2 is $\langle 1,0\rangle^{2^2-1}\langle 1,1\rangle\langle 0,0\rangle^{2^2-1}\langle 0,\text{-}1\rangle\langle 1,0\rangle^{2^3-1}\langle 1,1\rangle\langle 0,0\rangle^{2^4-1}\langle 0,\text{-}1\rangle\ldots$, and the trace of mean payoffs over r_2 has the set of accumulation points[3] $\{\langle x, 0\rangle | 1/3 \leq x \leq 2/3\}$.

Next, we define an MTMPBA which is a payoff system with two acceptance conditions F and G on Büchi fairness and mean payoffs, respectively. We capture a quantitative attribute of a run r via the set of accumulation points of the trace $mp_{\mathbf{w}}(r)$ of mean payoffs over r. Then, a mean-payoff acceptance condition G is given by a Boolean combination of the threshold conditions for the maximum or minimum value of the i-th projection of the set of accumulation points.

Definition 4. *An MTMPBA $\mathcal{A}$ is a tuple $\langle Q, \Sigma, \Delta, q_0, \mathbf{w}, F, G\rangle$ (or $\langle PS, F, G\rangle$ for a payoff system $PS = \langle Q, \Sigma, \Delta, q_0, \mathbf{w}\rangle$), where*

- *$F \subseteq Q$ is a* Büchi acceptance condition *given by a set of final states,*
- *$G : 2^{\mathbb{R}^d} \to \mathtt{Bool}$ is a* multi-threshold mean-payoff acceptance condition *such that $G(X)$ is a Boolean combination of* threshold conditions *of the form either $\max \pi_i(X) \sim c$ or $\min \pi_i(X) \sim c$ for $\sim \in \{<, >, \leq, \geq\}$, $c \in \mathbb{R}$ and the i-th projection π_i.*

The concepts of MTMPBAs are defined in a manner analogous to those of payoff systems. We denote by $Acc(\tau)$ the set of accumulation points of a trace τ. Note that, for an infinite run $r \in \mathrm{run}(\mathcal{A})$, the maximum (resp., minimum) of the set $\pi_i(Acc(mp_{\mathbf{w}}(r)))$ is equal to the limit supremum (resp., limit infimum) of the trace $mp_{\mathbf{w}[i]}(r)$. A threshold condition is *universal* if it has the form either $\max \pi_i(\cdot) < c$, $\max \pi_i(\cdot) \leq c$, $\min \pi_i(\cdot) > c$, or $\min \pi_i(\cdot) \geq c$; i.e., it asserts that "all" accumulation points of the i-th coordinate trace of mean payoffs over a run satisfy the inequality. Otherwise, it is *existential*; i.e., it asserts that "some" of the accumulation points satisfy the inequality.

An infinite run $r \in run(\mathcal{A})$ is *accepted* by $\mathcal{A}$ if both the Büchi acceptance condition F (i.e., a certain state $q \in F$ occurs infinitely often on r) and the mean-payoff acceptance condition $G(Acc(mp_{\mathbf{w}}(r)))$ hold. An infinite word $\sigma \in \Sigma^\omega$ is *accepted* by $\mathcal{A}$ if there exists a run r such that $\sigma = word(r)$ and r is accepted by $\mathcal{A}$ (i.e., $\mathcal{A}$ is an existential MTMPBA in a strict sense). A language $L \subseteq \Sigma^\omega$ (resp., an $\mathrm{LTL}^{\mathrm{mp}}$ sentence φ) is *recognized* by $\mathcal{A}$ if, for all $\sigma \in \Sigma^\omega$, σ is accepted by $\mathcal{A} \Leftrightarrow \sigma \in L$ (resp., $\sigma \models \varphi$). A language recognized by an MTMPBA with $\Delta : Q \times \Sigma \to Q$ and $F = Q$ is called a multi-threshold mean-payoff language [2]. Therefore, the class of languages recognized by MTMPBAs is the superclass of ω-regular languages and of multi-threshold mean-payoff languages.

[3] A point $\mathbf{x} \in \mathbb{R}^d$ is an accumulation point of a trace $\mathbf{x}_0\mathbf{x}_1\cdots \in (\mathbb{R}^d)^\omega$ if, for any open set containing $\mathbf{x}$, there are infinitely many indices $i_1, i_2, \ldots$ such that $\mathbf{x}_{i_1}, \mathbf{x}_{i_2}, \ldots$ belong to the open set.

Example 4. Consider the MTMPBAs $\mathcal{A}_1 = \langle PS, \{q_0\}, \min \pi_1(\cdot) \geq 1/2 \wedge \min \pi_2(\cdot) \geq 0\rangle$ and $\mathcal{A}_2 = \langle PS, \{q_0\}, \max \pi_1(\cdot) > 1/2 \wedge \min \pi_2(\cdot) < 0\rangle$, where PS is the payoff system of Example 3. Both runs r_1 and r_2 in Example 3 satisfy the Büchi condition $\{q_0\}$. The traces of mean payoffs over r_1 and r_2 have the respective sets $\{\langle 1, 1/2\rangle\}$ and $\{\langle x, 0\rangle | 1/3 \leq x \leq 2/3\}$ of accumulation points. Thus $\mathcal{A}_1$ accepts r_1, but rejects r_2, and $\mathcal{A}_2$ rejects both r_1 and r_2.

Regarding the closure properties of the class of languages recognized by MTMPBAs, the following theorem holds. (The proof is omitted from this paper.)

Theorem 5. *The class of languages recognized by MTMPBAs is closed under union and intersection.*

3.2 Emptiness Problems

An algorithm for the emptiness problems of multi-threshold mean-payoff languages has been proposed in [2]. An algorithm for the emptiness problems of MTMPBAs has also been proposed as a part of a procedure for the model-checking of $\mathrm{LTL}^{\mathrm{lim}}$ [8], and is based on the algorithm of [2]. The decision problems of $\mathrm{LTL}^{\mathrm{mp}}$ can be reduced to the emptiness problems of MTMPBAs (see Section 4), and hence can be decided by the algorithm of [8]. However, the complexity of that algorithm is exponential with respect to the size of the state space of the automaton.

In this paper, we reduce the emptiness problems of MTMPBAs to *linear constraint satisfaction problems* (LCSPs), which can be solved by *linear programming* (LP) methods. For an MTMPBA, the existence of an accepting run can be inferred from the existence of some sets of cyclic runs. Then, the solution of each LCSP is associated with a set of cyclic runs, and a set of solutions indicates the existence of an accepting run on the automaton.

Lemma 6. *Let $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, \mathbf{w}, F, G\rangle$ be a d-dimensional MTMPBA, where $G(\cdot) = \bigwedge_{1 \leq i \leq d} \min \pi_i(\cdot) \sim_i 0$. The following statements are equivalent.*

- *There exists an accepting run on $\mathcal{A}$.*
- *There exists a reachable (and maximal) SCC $\langle Q', \Delta'\rangle$ on $\mathcal{A}$ such that (i) $F \cap Q' \neq \emptyset$ and (ii) there exists a non-negative solution $\mathbf{x}$ for linear constraints (1)-(4), and the following conditions also hold:*

(ii-a) for each existential threshold condition of the form $\min \pi_i(\cdot) \leq 0$, there exists a non-negative solution $\mathbf{x}$ for linear constraints (1)-(4) and (5),

(ii-b) for each existential threshold condition of the form $\min \pi_i(\cdot) < 0$, there exists a non-negative solution $\mathbf{x}$ for linear constraints (1)-(4) and (6),

where $\mathbf{x}$ is a $|\Delta'|$-dimensional vector, x_δ is the element of the vector $\mathbf{x}$ associated with $\delta \in \Delta'$ and the linear constraints are:

$$\sum_{\delta \in \Delta'} x_\delta \geq 1, \tag{1}$$

$$\sum_{\delta \in \Delta' \text{ s.t. } post(\delta) = q} x_\delta = \sum_{\delta \in \Delta' \text{ s.t. } pre(\delta) = q} x_\delta \quad \text{for each } q \in Q', \tag{2}$$

$$\sum_{\delta \in \Delta'} \mathbf{w}[j](\delta) \cdot x_\delta \geq 0 \quad \text{for each } j \text{ such that } \sim_j \text{ is } \geq, \tag{3}$$

$$\sum_{\delta\in\Delta'} \mathbf{w}[j](\delta)\cdot x_\delta \;\geq\; 1 \quad \textit{for each } j \textit{ such that } \sim_j \textit{ is } >, \tag{4}$$
$$\sum_{\delta\in\Delta'} \mathbf{w}[i](\delta)\cdot x_\delta \;\leq\; 0, \tag{5}$$
$$\sum_{\delta\in\Delta'} \mathbf{w}[i](\delta)\cdot x_\delta \;\leq\; -1. \tag{6}$$

Proof. Let n be the number of existential threshold conditions in G, and fix a reachable SCC $S = \langle Q', \Delta' \rangle$ on $\mathcal{A}$.

First, consider a solution $\mathbf{x}$ for the linear constraints (1) and (2). If $\mathbf{x}$ is an integer vector, each variable x_δ can be interpreted as the number of occurrences of the transition δ on runs. With this interpretation, $\mathbf{x}$ implies the existence of m cyclic finite runs $r_1, \ldots, r_m \in \mathit{run}(\Delta')$. This is because the linear constraint (1) implies the existence of runs with positive length, and the linear constraint (2) implies that, for each state, the number of incoming transitions is equal to the number of outgoing transitions. Here, we shall denote by $WM_{\mathbf{x}}(\mathbf{w})$ the weighted mean $(\sum_{\delta\in\Delta'} x_\delta \cdot \mathbf{w}(\delta))/\sum_{\delta\in\Delta'} x_\delta$ of $\mathbf{w}$ with respect to $\mathbf{x}$ (in this sense, $\mathbf{x}$ and $\mathbf{w}$ are "weight" and "data" vectors, respectively). If $m = 1$, there exists a trivial run $r_0(r_1)^\omega \in \mathit{run}(\mathcal{A})$, since S is reachable. The trace $mp_{\mathbf{w}}(r_0(r_1)^\omega)$ of mean payoffs over this run converges on $WM_{\mathbf{x}}(\mathbf{w})$. It is equal to the mean payoff of r_1, and is independent of the prefix r_0. Otherwise, there exists a larger cyclic finite run of the form $r_1 r'_1 \cdots r_m r'_m \in \mathit{run}(\Delta')$, since S is a SCC. Then, we can obtain a run $r_0(r_1 r'_1 \cdots r_m r'_m)((r_1)^2 r'_1 \cdots (r_m)^2 r'_m)\cdots \in \mathit{run}(\mathcal{A})$. The trace of mean payoffs over the run also converges on $WM_{\mathbf{x}}(\mathbf{w})$ (i.e., the mean payoffs of $r_1, \ldots, r_m$). With this type of LCSP, given a solution $\mathbf{x}$ and a constant $c \geq 1$, the scalar product $c \cdot \mathbf{x}$ is also a solution. Therefore, even if $\mathbf{x}$ is a real vector, there still exists a run in $\mathit{run}(\mathcal{A})$ such that the ratio of the occurrence of transitions on r asymptotically approaches that of $\mathbf{x}$.

Next, consider a solution $\mathbf{x}$ of the linear constraints (1), (2) and $\sum_{\delta\in\Delta'} \mathbf{w}[k](\delta)\cdot x_\delta \geq 0$ (resp., $\sum_{\delta\in\Delta'} \mathbf{w}[k](\delta)\cdot x_\delta \geq 1$, $\sum_{\delta\in\Delta'} \mathbf{w}[k](\delta)\cdot x_\delta \leq 0$, and $\sum_{\delta\in\Delta'} \mathbf{w}[k](\delta)\cdot x_\delta \leq -1$). In a manner analogous to the first case, if a solution exists, there exists a run $r \in \mathit{run}(\mathcal{A})$ such that $\min \pi_k(\mathit{Acc}(mp_{\mathbf{w}}(r))) \sim_k 0$ holds, where $\sim_k$ is $\geq$ (resp., $>$, $\leq$ and $<$). This is because the k-th coordinate $WM_{\mathbf{x}}(\mathbf{w}[k])$ of the weighted mean with respect to $\mathbf{x}$ is greater than or equal to 0 (resp., greater than 0, less than or equal to 0, and less than 0). Otherwise, there is no run satisfying the threshold condition $\min \pi_k(\cdot) \sim_k 0$ on S. Hence, there exists a solution $\mathbf{x}$ of linear constraints (1)-(4) (and either (5) for $\min \pi_i(\cdot) \leq 0$ or (6) for $\min \pi_i(\cdot) < 0$) iff there exists a run $r_{\mathbf{x}} \in \mathit{run}(\mathcal{A})$ such that $r_{\mathbf{x}}$ satisfies all universal threshold conditions in G (and either $\min \pi_i(\cdot) \leq 0$ or $\min \pi_i(\cdot) < 0$).

Accordingly, if $n = 0$, the condition (ii) holds iff there exists a run satisfying G. Otherwise, the condition (ii) holds iff there exist runs $r_{\mathbf{x}_{\theta_1}}, \ldots, r_{\mathbf{x}_{\theta_n}} \in \mathit{run}(\mathcal{A})$ corresponding to solutions $\mathbf{x}_{\theta_1}, \ldots, \mathbf{x}_{\theta_n}$ for existential threshold conditions $\theta_1, \ldots, \theta_n$ in G. The trace of mean payoffs over $r_{\mathbf{x}_{\theta_k}}$ converges on the point $WM_{\mathbf{x}_{\theta_k}}(\mathbf{w})$, and $G(\{WM_{\mathbf{x}_{\theta_1}}(\mathbf{w}), \ldots, WM_{\mathbf{x}_{\theta_n}}(\mathbf{w})\})$ holds. This is because each of the runs satisfies all of the universal threshold conditions in G, and each of the existential threshold conditions is satisfied at least by one of the runs. Therefore, we can construct a run such that the trace of mean payoffs over the run comes arbitrarily close to every accumulation point $WM_{\mathbf{x}_{\theta_k}}(\mathbf{w})$ infinitely often. Consequently, the condition (ii) holds iff there exists a run satisfying G.

In addition, if such a run exists and condition (i) holds, there exists a run such that both F and G hold [8]. Hence, there exists an accepting run on $\mathcal{A}$ iff there exists a SCC on $\mathcal{A}$ satisfying the conditions (i) and (ii). □

Note that we can assume, without loss of generality, that (a) each coordinate is referred to by just one threshold condition, since the duplication of the coordinates of a weight function $\mathbf{w}$ does not change the recognizing language, and (b) a threshold condition has the form $\min \pi_i(\cdot) \sim 0$, since any threshold condition can be represented in this form via an affine transformation of $\mathbf{w}$.

Therefore, the emptiness problems of MTMPBAs can be reduced to LCSPs.

Theorem 7. *The emptiness problem of an MTMPBA is decidable in exponential time.*

Proof. Let $\mathcal{A} = \langle Q, \Sigma, \Delta, q_0, \mathbf{w}, F, G\rangle$ be an MTMPBA, G_i the i-th disjunct of a DNF formula $\bigvee G_i$ equivalent to G, and $\mathbf{w}_i$ the affine transformation of $\mathbf{w}$ for G_i, where each coordinate is referred to by just one threshold condition in G_i, and each G_i has the form $\bigwedge \min \pi_j(\cdot) \sim 0$.

The language recognized by $\mathcal{A}$ is empty iff the language recognized by the MTMPBA $\mathcal{A}_i = \langle Q, \Sigma, \Delta, q_0, \mathbf{w}_i, F, G_i\rangle$ is empty for all G_i. For G_i and reachable (and maximal) SCC S_{ik} on $\mathcal{A}_i$, each LCSP in Lemma 6 can be solved by LP methods in polynomial time for $|S_{ik}|$ and $|G|$. We must solve $\mathcal{O}(|G|)$ LCSPs to decide whether S_{ik} satisfies the condition (i) and (ii) in Lemma 6. Therefore, the emptiness problem of $\mathcal{A}_i$ can be solved in polynomial time for $|Q|$ and $|G|$.

In general, the number of disjuncts of a negation-free DNF formula equivalent to G is exponential in $|G|$. Hence, the complexity of the emptiness problem is polynomial in $|Q|$ and exponential in $|G|$. □

The algorithm of [8] computes explicitly a counterpart to the solution region for the linear constraints (1) and (2). The complexity is linear in the number of simple cyclic finite runs on an automaton, and that number is roughly estimated to be exponential in $|Q|$ [8]. In comparison, our algorithm captures the region implicitly, and solutions can be found in polynomial time for $|Q|$.

Example 5. Consider the MTMPBA $\mathcal{A}'_2 = \langle Q, 2^{\{p_0,p_1\}}, \Delta, q_0, \mathbf{w}', \{q_0\}, \min \pi_1(\cdot) < 0 \wedge \min \pi_2(\cdot) < 0\rangle$, obtained by affine transformation of the MTMPBA $\mathcal{A}_2 = \langle Q, 2^{\{p_0,p_1\}}, \Delta, q_0, \mathbf{w}, \{q_0\}, \max \pi_1(\cdot) > 1/2 \wedge \min \pi_2(\cdot) < 0\rangle$ of Example 4, where $\mathbf{w}'[1](\delta) = -\mathbf{w}[1](\delta) + 1/2$ and $\mathbf{w}'[2](\delta) = \mathbf{w}[2](\delta)$ for $\delta \in \Delta$. Trivially, the SCC $\langle Q, \Delta\rangle$ is reachable and maximal, and has a final state q_0. $\mathcal{A}'_2$ has two existential threshold conditions, and hence we must solve two LCSPs to decide whether $\mathcal{A}'_2$ (and also $\mathcal{A}_2$) is empty. For a non-negative vector $\langle x_{\delta_1}, \ldots, x_{\delta_6}\rangle$, the linear constraints are: (1) $\sum_{k=1}^{6} x_{\delta_k} \geq 1$, (2) $x_{\delta_1} = x_{\delta_3}$, $x_{\delta_1} + x_{\delta_4} = x_{\delta_3} + x_{\delta_6}$ and $x_{\delta_4} = x_{\delta_6}$, (6-1) $-\frac{1}{2}(x_{\delta_1} + x_{\delta_2} + x_{\delta_3}) + \frac{1}{2}(x_{\delta_4} + x_{\delta_5} + x_{\delta_6}) \leq -1$ for $\min \pi_1(\cdot) < 0$, and (6-2) $x_{\delta_3} - x_{\delta_6} \leq -1$ for $\min \pi_2(\cdot) < 0$. As a result, $\langle 1, 0, 1, 0, 0, 0\rangle$ and $\langle 0, 0, 0, 1, 0, 1\rangle$ (for example) turn out to be solutions for {(1), (2) and (6-1)} and {(1), (2) and (6-2)}, respectively. These vectors are indicative of the sets $\{\delta_1\delta_3\}$ and $\{\delta_4\delta_6\}$ of single cyclic run. Therefore, we can construct an accepting run on $\mathcal{A}_2$; e.g., $(\delta_1\delta_3)^{2^2}(\delta_4\delta_6)^{2^{2^2}}(\delta_1\delta_3)^{2^{2^3}}(\delta_4\delta_6)^{2^{2^4}}\ldots$.

4 Decision and Optimization Problems of LTL$^{\mathrm{mp}}$

In this section, we present algorithms for the decision and optimization problems of LTL$^{\mathrm{mp}}$.

In a manner analogous to the decision problems of classical LTL, we can reduce the decision problems of LTL$^{\mathrm{mp}}$ to the emptiness problems of automata, which recognize LTL$^{\mathrm{mp}}$ formulae. First, we show how to construct such an automaton from a given LTL$^{\mathrm{mp}}$ sentence.

Lemma 8. *For an LTL^{mp} sentence φ, there exists an MTMPBA recognizing φ.*

Proof. Consider a *future-independent* payment t of the form $\sum c_i \cdot \mathbf{1}_{\psi_i}$ (i.e., temporal operators do not appear in every ψ_i). We can easily translate t into an alphabetic weight function $\mathbf{w}_t(\delta) = \sum_{letter(\delta) \text{ satisfies } \psi_i} c_i$, for $\delta \in \Delta$. Thus, the MTMPBA $\mathcal{A} = \langle \{q_0\}, 2^{AP}, \{q_0\} \times 2^{AP} \times \{q_0\}, q_0, \mathbf{w}_t, \{q_0\}, \min \pi_1(\cdot) > c \rangle$ recognizes the formula $\underline{\mathsf{MP}}(t) > c$. For a simple mean-payoff formula for such a payment in another form, we can construct a recognizing MTMPBA in a similar way. Therefore, we can construct an MTMPBA recognizing a given LTL$^{\mathrm{mp}}$ sentence φ if φ has no future-dependent variable, since any LTL$^{\mathrm{mp}}$ formula can be represented in MPNF and Theorem 5 holds. The size of an MPNF formula equivalent to φ is at worst linear in $|\varphi|$ and exponential in the number n of mean-payoff formulae in φ. Hence, the size of the resulting automaton is at worst exponential in $|\varphi|$ and n.

Next, consider an LTL$^{\mathrm{mp}}$ sentence φ with m future-dependent characteristic variables $\mathbf{1}_{\psi_1}, \ldots, \mathbf{1}_{\psi_m}$. Then, we can obtain another LTL$^{\mathrm{mp}}$ sentence φ', which has fresh predictive propositions $p_1, \ldots, p_m$ as follows:

$$\varphi' = \varphi[\psi_1, \ldots, \psi_m / p_1, \ldots, p_m] \wedge \textstyle\bigwedge_{1 \leq j \leq m} \Box(p_j \leftrightarrow \psi_j).$$

This sentence φ' has no future-dependent variable, and preserves the behavioral characteristics represented by φ. Therefore, we can obtain an MTMPBA $\mathcal{A}_\varphi$ that recognizes φ, by eliminating $p_1, \ldots, p_m$ from an MTMPBA $\mathcal{A}_{\varphi'}$ that recognizes φ'. The size of the resulting automaton is at worst exponential in also m. □

Example 6. Consider the following LTL$^{\mathrm{mp}}$ formulae $\varphi_1, \ldots, \varphi_4$: $\varphi_1 = \neg p_0 \wedge \neg p_1$, $\varphi_2 = \Box((\neg p_0 \vee \neg p_1) \wedge ((p_0 \vee p_1) \rightarrow \mathbf{X}(\neg p_0 \wedge \neg p_1)))$, $\varphi_3 = \Box\Diamond(p_0 \vee p_1)$ and $\varphi_4 = \underline{\mathsf{MP}}(\mathbf{1}_{\psi_1}) \geq 1/2 \wedge \underline{\mathsf{MP}}(\mathbf{1}_{p_0} - \mathbf{1}_{p_1}) \geq 0$, where $\psi_1 = (\neg p_0 \wedge \neg p_1)\mathbf{U}p_1$. The MTMPBA $\mathcal{A}_1$ in Example 4 recognizes the LTL$^{\mathrm{mp}}$ sentence $\bigwedge_{1 \leq i \leq 4} \varphi_i$, and $\mathcal{A}_1$ or an MTMPBA equivalent to it can easily be obtained from the sentence. Intuitively, φ_1 represents the outgoing transitions $\langle q_0, A, q_1 \rangle$ and $\langle q_0, A, q_2 \rangle$ from the initial state q_0 of $\mathcal{A}_1$, φ_2 represents the transition relation of $\mathcal{A}_1$, and φ_3 and φ_4 represent the Büchi and mean-payoff acceptance conditions of $\mathcal{A}_1$, respectively. The nondeterminism of the transitions $\langle q_0, A, q_1 \rangle$ and $\langle q_0, A, q_2 \rangle$ on $\mathcal{A}_1$ is caused by the future-dependent payment $\mathbf{1}_{\psi_1}$.[4]

[4] Even if we consider *multi-threshold mean-payoff "Rabin" automata*, there is no deterministic automaton that recognizes the sentence $\bigwedge_{1 \leq i \leq 4} \varphi_i$. (The proof of this fact is omitted from this paper.) However, some future-dependent payments (e.g., $\mathbf{1}_{\mathbf{X}p}$ for $p \in AP$) do not exhibit this result.

In a manner analogous to the classical LTL model-checking, the model-checking of an $\mathrm{LTL}^{\mathrm{mp}}$ formula φ against a payoff system PS can be reduced to the emptiness problem of a synchronized product of PS and an automaton recognizing $\neg\varphi$, considering the proper variable assignment. In this paper, we define the satisfaction relation between a payoff system and an $\mathrm{LTL}^{\mathrm{mp}}$ formula as follows.

Definition 9 ($PS \models \varphi$)**.** *Let $PS = \langle Q, 2^{AP}, \Delta, q_0, \mathbf{w}\rangle$ be a d-dimensional payoff system, and let φ be an LTL^{mp} formula over a set V of free variables, where the free variables are indexed and $|V| = d$. In addition, we assume that the i-th free variable v_i is associated with the i-th coordinate of $\mathbf{w}$; i.e., we employ the assignment $\alpha_{\mathbf{w},r}$ such that $\alpha_{\mathbf{w},r}(v_i) = \mathit{payoff}_{\mathbf{w}[i]}(r)$ for an infinite run r on PS.*

We define the satisfaction relation $PS \models \varphi$ by $\mathit{word}(r), \alpha_{\mathbf{w},r} \models \varphi$ for all $r \in \mathit{run}(PS)$.

A little trick is required to assign traces of payoffs over a run of PS to free variables in φ on the synchronized product. Then, we reduce the model-checking to the emptiness problem.

Theorem 10. *The model-checking for an LTL^{mp} specification against a payoff system is decidable in exponential time.*

Proof. Let $PS = \langle Q, 2^{AP}, \Delta, q_0, \mathbf{w}\rangle$ be a payoff system, φ an $\mathrm{LTL}^{\mathrm{mp}}$ formula over a set V of free variables, φ_i the i-th disjunct of an MPNF formula $\bigvee \varphi_i$ equivalent to $\neg\varphi$, ψ_{ij} the j-th simple mean-payoff formula in φ_i, $\sum(c_{ijk} \cdot \mathbf{1}_{\varphi_{ijk}} \cdot \prod v_{ijkl})$ the payment for ψ_{ij}, $\mathbf{w}[ijkl]$ a coordinate function of $\mathbf{w}$ associated with the free variable v_{ijkl}, and n the number of mean-payoff formulae in φ.

$PS \models \varphi$ iff the language L_{PS,φ_i} is empty for all φ_i, where L_{PS,φ_i} is a set of words over runs $r \in \mathit{run}(PS)$ such that $\mathit{word}(r)$ satisfies φ_i under the assignment of each trace $\mathit{payoff}_{\mathbf{w}[ijkl]}(r)$ of payoffs over r to the corresponding free variable v_{ijkl} in φ_i. Therefore, we construct MTMPBAs recognizing such languages, and decide whether $PS \models \varphi$ by checking the emptiness of them.

Then, we show how to construct such MTMPBAs. First, we construct an MTMPBA $\mathcal{A}_{\varphi_i'} = \langle Q_i, 2^{AP}, \Delta_i, q_{i0}, \mathbf{w}_i, F_i, G_i\rangle$ that recognize $\varphi_i' = \varphi_i[v/0$ for all free variable $v \in V]$. We assume that the j-th coordinate function $\mathbf{w}_i[j]$ of $\mathbf{w}_i$ is associated with the payment for ψ_{ij}, and predictive propositions for future-dependent characteristic variables are still annotated on the automaton. Next, we construct a synchronized product $PS \otimes \mathcal{A}_{\varphi_i} = \langle Q \times Q_i, 2^{AP}, \Delta_i', \langle q_0, q_{i0}\rangle, \mathbf{w}_i', Q \times F_i, G_i\rangle$ of PS and $\mathcal{A}_{\varphi_i}$, considering the proper variable assignment as follows:

$$\Delta_i' = \{\langle\langle q_1, q_1'\rangle, a, \langle q_2, q_2'\rangle\rangle \mid \langle q_1, a, q_2\rangle \in \Delta, \langle q_1', a, q_2'\rangle \in \Delta_i\},$$
$$\mathbf{w}_i'[j](\langle\langle q_1, q_1'\rangle, a, \langle q_2, q_2'\rangle\rangle) = \mathbf{w}_i[j](\langle q_1', a, q_2'\rangle) + \textstyle\sum_{a \text{ satisfies } \varphi_{ijk}} (c_{ijk} \cdot \prod \mathbf{w}[ijkl](\langle q_1, a, q_2\rangle)).$$

We use annotated predictive propositions to check whether the letter a satisfies φ_{ijk} if φ_{ijk} has temporal operators. The automaton $PS \otimes \mathcal{A}_{\varphi_i}$ recognizes L_{PS,φ_i}.

The automaton $PS \otimes \mathcal{A}_{\varphi_i}$ has $|Q| \cdot |Q_i|$ states and a conjunctive mean-payoff acceptance condition G_i, where $|Q_i| = \mathcal{O}(2^{|\varphi|})$ and $|G_i| = \mathcal{O}(n)$. The emptiness

problem for $PS \otimes \mathcal{A}_{\varphi_i}$ can be solved in polynomial time for $|Q| \cdot |Q_i|$ and $|G_i|$, since G_i is conjunctive (Theorem 7). The number of disjuncts of an MPNF formula equivalent to $\neg\varphi$ is exponential in n, and hence the complexity of the model-checking is polynomial in $|Q|$ and exponential in $|\varphi|$ and n. □

Our algorithm can accomplish the model-checking of LTL$^{\mathrm{mp}}$ with much less complexity than the algorithm of [8], which is roughly estimated to be exponential in $|Q|$, doubly exponential in $|\varphi|$, and triply exponential in n.

In a manner analogous to the classical LTL satisfiability problem, we can reduce the satisfiability problem for an LTL$^{\mathrm{mp}}$ sentence φ to the non-emptiness problem of an automaton recognizing φ.

Theorem 11. *The satisfiability problem of an LTL^{mp} sentence is decidable in exponential time.*

Proof. An LTL$^{\mathrm{mp}}$ sentence φ with n mean-payoff formulae is satisfiable iff an MTMPBA that recognizes φ is not empty. By Theorem 7 and Lemma 8, the satisfiability problem is decidable in exponential time for $|\varphi|$ and n. □

We can eventually reduce the satisfiability problem of LTL$^{\mathrm{mp}}$ to LCSPs, which can be solved by LP methods. Therefore, some optimization problems of LTL$^{\mathrm{mp}}$ can be also solved by LP methods.

Theorem 12. *The maximization/minimization problem for a mean-payoff objective ($\overline{\mathsf{MP}}(t)$ or $\underline{\mathsf{MP}}(t)$), which is subject to an LTL^{mp} sentence, is solvable in exponential time.*

Proof. Let θ be an objective ($\overline{\mathsf{MP}}(t)$ or $\underline{\mathsf{MP}}(t)$), φ an LTL$^{\mathrm{mp}}$ sentence, φ_i the i-th disjunct MPNF formula $\bigvee \varphi_i$ equivalent to φ, and n the number of mean-payoff formulae in φ.

The optimal value for θ, which is subject to φ, can be obtained as the optimal value in the set of values $opt_\theta(\varphi_i)$, where $opt_\theta(\varphi_i)$ is the optimal value for θ, which is subject to φ_i.

Such $opt_\theta(\varphi_i)$ can be found by using an MTMPBA $\mathcal{A}^t_{\varphi_i}$, which recognizes φ_i and has an additional coordinate associated with t. For a disjunct φ_i, $\mathcal{A}^t_{\varphi_i}$ can be obtained by a construction similar to that used in Lemma 8. Let w_t be the weight function of this additional coordinate, and let G_i be a mean-payoff acceptance condition of $\mathcal{A}^t_{\varphi_i}$, where G_i has the form $\bigwedge \min \pi_j(\cdot) \sim 0$ and m existential threshold conditions $p_{i1}, \ldots, p_{im}$ $(m \leq |G_i| \leq n)$.

Then, $opt_\theta(\varphi_i)$ is obtained as the optimal value in a set of values $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ for reachable SCC S_{ik} on $\mathcal{A}^t_{\varphi_i}$, where S_{ik} satisfies the condition (i) and (ii) in Lemma 6 and $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ is the optimal value for θ, subject to φ_i on S_{ik}.

If S_{ik} satisfies the condition (i) and (ii), we obtain $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ as follows:

– If (a) $m = 0$ or (b) the problem is the maximization of $\overline{\mathsf{MP}}(t)$ or the minimization of $\underline{\mathsf{MP}}(t)$, $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ is the maximized/minimized weighted mean $WM_{\mathbf{x}}(w_t)$ of w_t with respect to the solution $\mathbf{x}$, where $\mathbf{x}$ is subject to all universal threshold conditions in G_i on S_{ik} (i.e., the linear constraints (1)-(4) in Lemma 6).

– Otherwise, $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ is the minimum/maximum value in the set of the maximized/minimized weighted means $WM_{\mathbf{x}_{ik1}}(w_t), \ldots, WM_{\mathbf{x}_{ikm}}(w_t)$ of w_t with respect to the solutions $\mathbf{x}_{ik1}, \ldots, \mathbf{x}_{ikm}$, where each $\mathbf{x}_{ikl}$ is subject to all universal threshold conditions in G_i and p_{il} on S_{ik} (i.e., the linear constraints (1)-(4), and either (5) or (6) depending on p_{il} in Lemma 6).

If (a) holds, $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ is trivially optimal for φ_i on S_{ik}. Otherwise, note that each p_{il} asserts that "some" accumulation points of mean payoffs over a run satisfy the inequality. Therefore, if (b) holds, $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ is obtained independently from $p_{i1}, \ldots, p_{im}$. Otherwise, $opt_\theta(\mathcal{A}^t_{\varphi_i}, S_{ik})$ is given by the minimum/maximum value in the set of the maximized/minimized weighted means $WM_{\mathbf{x}_{ik1}}(w_t), \ldots, WM_{\mathbf{x}_{ikm}}(w_t)$ for $p_{i1}, \ldots, p_{im}$. Each optimization problem for the weighted mean is a linear fractional programming problem (LFPP), which can be solved by LP methods in polynomial time for $|S_{ik}|$ and n.

The basic flow of the optimization on $\mathcal{A}^t_{\varphi_i}$ is similar to that of the emptiness problem of an MTMPBA $\mathcal{A}_{\varphi_i}$ recognizing φ_i. Instead of each LCSP in the emptiness problem, we solve the corresponding LFPP in the optimization. Hence, the complexity of the optimization problem is exponential in $|\varphi|$ and n. □

Therefore, we can analyze performance limitations under given qualitative and quantitative specifications described in $\mathrm{LTL}^{\mathrm{mp}}$. LP and related techniques can be also applied to other optimization problems (e.g., maximization/minimization problems for the limit supremum or limit infimum of the ratio of the mean payoffs for two payments) and multi-objective optimization problems, as in [14]. We conjecture that this LP-based approach for specification optimization can effectively be applied to optimal synthesis for temporal logic specifications.

5 Conclusions and Future Work

In this paper, we introduced $\mathrm{LTL}^{\mathrm{mp}}$, which is an extension of LTL with mean-payoff formulae. A mean-payoff formula is a threshold condition for the limit supremum or limit infimum of the mean payoffs pertaining to a given payment. This extension allows us to describe specifications that reflect qualitative and quantitative requirements on long-run average costs and frequencies of satisfying temporal properties. Moreover, we introduced multi-threshold mean-payoff Büchi automata (MTMPBAs), which are payoff automata with Büchi acceptance conditions and multi-threshold mean-payoff acceptance conditions. Then, we developed an algorithm for solving the emptiness problems of MTMPBAs, by reducing the problems to linear constraint satisfaction problems. The decision problems of the logic can be reduced to the emptiness problems, and hence we obtained exponential-time algorithms for model- and satisfiability-checking of the logic. An additional advantage of the reduction is that some optimization problems for specifications described in the logic can be solved by linear programming methods in exponential time.

Future work will be devoted to a detailed analysis of the determinizability of automata that recognize sentences described in mean-payoff extensions of LTL

and to developing the realizability-checking and quantitative synthesis methods of the extensions.

References

1. Acacia+, http://lit2.ulb.ac.be/acaciaplus/
2. Alur, R., Degorre, A., Maler, O., Weiss, G.: On Omega-Languages Defined by Mean-Payoff Conditions. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 333–347. Springer, Heidelberg (2009)
3. Andova, S., Hermanns, H., Katoen, J.P.: Discrete-Time Rewards Model-Checked. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 88–104. Springer, Heidelberg (2004)
4. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Verifying Continuous Time Markov Chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)
5. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time markov chains. IEEE Transactions on Software Engineering 29(6), 524–541 (2003)
6. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: On the Logical Characterisation of Performability Properties. In: Welzl, E., Montanari, U., Rolim, J.D.P. (eds.) ICALP 2000. LNCS, vol. 1853, pp. 780–792. Springer, Heidelberg (2000)
7. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better Quality in Synthesis through Quantitative Objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
8. Boker, U., Chatterjee, K., Henzinger, T., Kupferman, O.: Temporal specifications with accumulative values. In: LICS 2011, pp. 43–52 (2011)
9. Brázdil, T., Forejt, V., Kretínský, J., Kucera, A.: The satisfiability problem for probabilistic ctl. In: LICS 2008, pp. 391–402 (2008)
10. Černý, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative Synthesis for Concurrent Programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 243–259. Springer, Heidelberg (2011)
11. Chatterjee, K., Doyen, L., Edelsbrunner, H., Henzinger, T.A., Rannou, P.: Mean-Payoff Automaton Expressions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 269–283. Springer, Heidelberg (2010)
12. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative Languages. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 385–400. Springer, Heidelberg (2008)
13. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and Synthesizing Systems in Probabilistic Environments. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
14. Chatterjee, K., Majumdar, R., Henzinger, T.A.: Markov Decision Processes with Multiple Objectives. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 325–336. Springer, Heidelberg (2006)
15. Droste, M., Gastin, P.: Weighted automata and weighted logics. Theoretical Computer Science 380(1-2), 69–86 (2007)
16. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing 6(5), 512–535 (1994)

17. Kupferman, O., Lustig, Y.: Lattice Automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
18. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL 1989, pp. 179–190 (1989)
19. Pnueli, A.: The temporal logic of programs. In: FOCS 1977, pp. 46–57 (1977)
20. PRISM, http://www.prismmodelchecker.org/
21. SPIN, http://spinroot.com/spin/
22. Tomita, T., Hagihara, S., Yonezaki, N.: A probabilistic temporal logic with frequency operators and its model checking. In: INFINITY 2011. EPTCS, vol. 73, pp. 79–93 (2011)

Time Constraints with Temporal Logic Programming*

Meng Han, Zhenhua Duan**, and Xiaobing Wang

Institute of Computing Theory and Technology, and ISN Laboratory
Xidian University, Xi'an 710071, P.R. China
xdhanmeng@163.com, {zhhduan,xbwang}@mail.xidian.edu.cn

Abstract. This paper presents an approach for the real-time extension of Projection Temporal Logic (PTL) and the corresponding programming language, Timed Modeling, Simulation and Verification Language (TMSVL). To this end, quantitative temporal constraints are employed to limit the time duration bounded on a formula or a program. First, the syntax and semantics of TPTL formulas are defined and some logic laws are given. Then, the corresponding executable programming language TMSVL is presented. Moreover, the operational semantics of TMSVL is formalized. Finally, an example of modeling and verification is given to show how TMSVL works.

Keywords: temporal logic, real-time system, programming language, modeling, verification.

1 Introduction

Real-time systems play an important role in our modern society. They are used in many safety critical systems such as aircrafts, traffic controllers, military commands and control systems and so on. It is critical to ensure the correctness of such systems. Testing is an effective means for ensuring high reliable software quality in engineering practice. However, real-time systems always depend on external environment with many uncertainties existing. Therefore, it is difficult to ensure the correctness and reliability of safety critical real-time systems. Alternatively, formal engineering method is a combination of strictly mathematical approaches and practical engineering approaches which can be used to model, simulate and verify real-time systems.

Temporal logic (TL) was proposed for the purpose of specification and verification of concurrent systems, and had been widely applied in verifying systems of software engineering and digital circuits. Projection Temporal Logic (PTL) [6,7,9] is an interval-based temporal logic, which is a useful formalism for reasoning about period of time with hardware and software systems. The Modeling,

* This research is supported by the NSFC Grant No. 61003078, 61133001, 60910004, and 973 Program Grant No. 2010CB328102, and ISN Lab Grant No. ISN1102001.
** Corresponding author.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 266–282, 2012.

Simulation, and Verification Language (MSVL)[10,11,9] is an executable subset of PTL and can be used to model, simulate and verify concurrent systems. To do so, a system is modeled by an MSVL program and a property of the system is specified by a Propositional Projection Temporal Logic (PPTL) formula. Thus, whether or not the system satisfies the property can be verified with the same logic framework. However, since PTL has no metric for describing absolute time, only the specification of qualitative temporal requirements is allowed. Thus, MSVL could not be used to model a realistic real-time system with some circumstances. Therefore, we are motivated to extend PTL and MSVL so that absolute time constraints can be used for describing real-time systems.

The main contributions of this paper are as follows: 1. a time duration is defined to constraint a formula in a time interval, which is based on an explicit time variable; then, PTL is extended to timed PTL (TPTL) with the time duration; 2. the programming language Timed MSVL (TMSVL) based on TPTL is developed, which can be used for modeling, simulation, and verification of real-time systems; 3. the operational semantics of TMSVL is also formalized to strictly describe the meaning of TMSVL programs; based on the operational semantics, the interpreter for TMSVL has been developed recently.

The rest of the paper is organized as follows. In the next section, Projection Temporal Logic and the programming language MSVL are briefly introduced. In section 3, the syntax and semantics of TPTL are presented and some derived formulas and logic laws are also provided. The programming language TMSVL and its operational semantics are defined in section 4. In section 5, as a case study, a video on demand application of real-time systems is modeled and some properties of the system are verified using TMSVL in detail. Some related works are compared in section 6. Finally, conclusions are drawn in section 7.

2 Preliminaries

2.1 PTL

Syntax. Let Π be a countable set of propositions, and V be a countable set of typed static and dynamic variables. Terms e and formulas p of the logic are defined as follows:

$$e ::= v \mid \bigcirc e \mid \ominus e \mid \mathsf{beg}(e) \mid \mathsf{end}(e) \mid f(e_1, \ldots, e_m)$$
$$p ::= \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_m) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \bigcirc p \mid (p_1, \ldots, p_m)\ \mathsf{prj}\ p \mid (p_1, \ldots, (p_i, \ldots, p_l)^{\oplus}, \ldots, p_m)\ \mathsf{prj}\ p$$

where v is a static or dynamic variable, and π is a proposition. In $f(e_1, \ldots, e_m)$ and $P(e_1, \ldots, e_m)$, where f is a function and P is a predicate. It is assumed that the types of the terms are compatible with those of the arguments of f and P. A formula (term) is called a state formula (term) if it does not contain any temporal operators (i.e., $\bigcirc$, $\ominus$ and prj), otherwise it is a temporal formula (term).

Semantics. A state s is a pair of assignments (I_{var}, I_{prop}) which, for each variable $v \in V$ gives $s[v] = I_{var}[v]$, and for each proposition $\pi \in \Pi$ gives

$s[\pi] = I_{prop}[\pi]$. Each $I_{var}[v]$ is a value of the appropriate type or *nil* (undefined), whereas $I_{prop}[\pi]$ is true or false. An interval $\sigma = \langle s_0, s_1, \ldots \rangle$ is a non-empty (possibly infinite) sequence of states. It is assumed that each static variable is assigned the same value in all the states over σ. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of the states in σ minus one. The concatenation of a finite interval $\sigma = \langle s_0, \ldots, s_t \rangle$ with another interval $\sigma' = \langle s'_0, \ldots \rangle$ (may be infinite) is denoted by $\sigma \cdot \sigma'$ and $\sigma \cdot \sigma' = \langle s_0, \ldots, s_t, s'_0, \ldots \rangle$. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set $\mathbb{N}_0$ of non-negative integers with added ω, $\mathbb{N}_\omega = \mathbb{N}_0 \cup \{\omega\}$, and extend the standard arithmetic comparison operators ($=$, $<$ and $\leq$) to $\mathbb{N}_\omega$, by setting $\omega = \omega$ and $n < \omega$, for all $n \in \mathbb{N}_0$. Furthermore, we define $\preceq$ as $\leq - \{(\omega, \omega)\}$. To simplify definitions, we will denote σ as $\langle s_0, \ldots, s_{|\sigma|} \rangle$, where $s_{|\sigma|}$ is undefined if σ is infinite. With such a notation, $\sigma_{(i..j)}$ (for $0 \leq i \preceq j \leq |\sigma|$) denotes the sub-interval $\langle s_i, \ldots, s_j \rangle$ and $\sigma^{(k)}$ (for $0 \leq k \preceq |\sigma|$) denotes $\langle s_k, \ldots, s_{|\sigma|} \rangle$.

To define the semantics of the projection operator we need an auxiliary operator. Let $\sigma = \langle s_0, s_1, \ldots \rangle$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \leq |\sigma|$. $\sigma{\downarrow}(r_1, \ldots, r_h) = \langle s_{t_1}, s_{t_2} \ldots, s_{t_l} \rangle, (t_1 < t_2 < \ldots < t_l)$. The projection of σ onto $r_1, \ldots, r_h$ is the interval, called projected interval, where $t_1, \ldots, t_l$ are obtained from $r_1, \ldots, r_h$ by deleting all duplicates. In other words, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. For example $\langle s_0, s_1, s_2, s_3, s_4, s_5 \rangle{\downarrow}(0, 2, 2, 2, 3, 4, 4, 5) = \langle s_0, s_2, s_3, s_4, s_5 \rangle$.

An interpretation for a PTL term or formula is a tuple $\mathfrak{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \ldots \rangle$ is an interval and $i, j, k \in \mathbb{N}_\omega$ are integers such that $i \leq k \preceq j \leq |\sigma|$. Intuitively, we use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ with the current state being s_k. Then, for every term e, the evaluation of e relative to $\mathfrak{I}$, denoted by $\mathfrak{I}[e]$, is defined by induction on terms in the following way:

$$
\begin{array}{ll}
\mathfrak{I}[v] & = s_k[v] \;=\; I^k_{var}[v] \text{ if } v \text{ is a variable} \\
\mathfrak{I}[f(e_1, \ldots, e_m)] & = \begin{cases} \mathfrak{I}[f](\mathfrak{I}[e_1], \ldots, \mathfrak{I}[e_m]) & \text{if } \mathfrak{I}[e_h] \neq nil \text{ for all } 1 \leq h \leq m \\ nil & \text{otherwise} \end{cases} \\
\mathfrak{I}[\bigcirc e] & = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases} \\
\mathfrak{I}[\ominus e] & = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases} \\
\mathfrak{I}[\mathsf{beg}(e)] & = (\sigma, i, i, j)[e] \\
\mathfrak{I}[\mathsf{end}(e)] & = \begin{cases} (\sigma, i, j, j)[e] & \text{if } j \neq \omega \\ nil & \text{otherwise} \end{cases}
\end{array}
$$

The satisfaction relation for formulas, $\models$, is inductively defined as follows.

1. $\mathfrak{I} \models \pi$ iff $s_k[\pi] = I^k_{prop}[\pi] = \mathsf{true}$.
2. $\mathfrak{I} \models P(e_1, \ldots, e_m)$ iff and $\mathfrak{I}[e_h] \neq nil$ $\mathfrak{I}[P](\mathfrak{I}[e_1], \ldots, \mathfrak{I}[e_m]) = \mathsf{true}$, for all $1 \leq h \leq m$.
3. $\mathfrak{I} \models e_1 = e_2$ iff $\mathfrak{I}[e_1] = \mathfrak{I}[e_2]$.
4. $\mathfrak{I} \models \neg p$ iff $\mathfrak{I} \not\models p$.
5. $\mathfrak{I} \models p \wedge q$ iff $\mathfrak{I} \models p$ and $\mathfrak{I} \models q$.
6. $\mathfrak{I} \models \bigcirc p$ iff $k < j$ and $(\sigma, i, k+1, j) \models p$.

7. $\Im \models \exists v : p$ iff $(\sigma', i, k, j) \models p$ for some $\sigma' \stackrel{v}{=} \sigma$.
8. $\Im \models (p_1, \ldots, p_m)$ prj p iff there are $k = r_0 \leq r_1 \leq \ldots \leq r_m \leq j$ such that $(\sigma, i, r_0, r_1) \models p_1$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ for all $1 < l \leq m$ and $(\sigma', 0, 0, |\sigma'|) \models p$ for σ' given by :
   - if $r_m < j$ then $\sigma' = \sigma\downarrow(r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$
   - if $r_m = j$ then $\sigma' = \sigma\downarrow(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.
9. $\Im \models (p_1, \ldots, (p_i, \ldots, p_l)^{\oplus}, \ldots, p_m)$ prj p iff one of following cases holds:
   - $1 \leq i \leq l \leq m$ and there exists an integer $n \geq 1$ and $\Im \models (p_1, \ldots, (p_i, \ldots, p_l)^{(n)}, \ldots, p_m)$ prj p, or
   - $1 \leq i \leq l = m, j = w$ and there exist infinitely many integers $k = r_0 \leq r_1 \leq \ldots \leq r_k \preceq \omega$ and $\lim_{k\to\infty} r_k = \omega$ such that $(\sigma, i, r_0, r_1) \models p_1$ and $(\sigma, r_{x-1}, r_{x-1}, r_x) \models p_x$ for all $1 \leq x \leq i-1$ and $(\sigma, r_{i+t(l-i+1)+n-1}, r_{i+t(l-i+1)+n-1}, r_{i+t(l-i+1)+n}) \models p_{i+n}$, for all $t \geq 0$ and $0 \leq n \leq l-i$, and $\sigma\downarrow(r_0, r_1, \ldots, r_h, \omega) \models p$ for some $h \in N_\omega$.

A formula P is satisfied by an interval σ, denoted by $\sigma \models P$, if $(\sigma, 0, 0, |\sigma|) \models P$. A formula P is called satisfiable if $\sigma \models P$ for some σ. A formula P is valid, denoted by $\models P$, if $\sigma \models P$ for all σ. A formula p is equivalent to another formula q, denoted by $p \equiv q$, if $\models \Box(p \leftrightarrow q)$.

The abbreviations true, false, $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $\mathsf{true} \equiv p \vee \neg p$ and $\mathsf{false} \equiv p \wedge \neg p$ for any formula p. We also use the following abbreviations:

$$\mathsf{empty} \stackrel{\text{def}}{=} \neg \bigcirc \mathsf{true} \qquad \mathsf{skip} \stackrel{\text{def}}{=} \bigcirc\mathsf{empty} \qquad p\,;\, q \stackrel{\text{def}}{=} (p, q)\ \mathsf{prj}\ \mathsf{empty}$$
$$\Diamond p \stackrel{\text{def}}{=} \mathsf{true}\,;\, p \qquad \odot p \stackrel{\text{def}}{=} \mathsf{empty} \vee \bigcirc p \qquad \Box p \stackrel{\text{def}}{=} \neg\Diamond\neg p$$
$$\mathsf{more} \stackrel{\text{def}}{=} \neg\mathsf{empty} \qquad x := e \stackrel{\text{def}}{=} \bigcirc x = e \wedge \mathsf{skip} \qquad \mathsf{fin}(p) \stackrel{\text{def}}{=} \Box(\mathsf{empty} \rightarrow p)$$
$$p^+ \stackrel{\text{def}}{=} (p^{\oplus})\ \mathsf{prj}\ \mathsf{empty} \qquad p^* \stackrel{\text{def}}{=} (p^{\circledast})\ \mathsf{prj}\ \mathsf{empty} \qquad \mathsf{halt}(p) \stackrel{\text{def}}{=} \Box(\mathsf{empty} \leftrightarrow p)$$
$$\mathsf{keep}(p) \stackrel{\text{def}}{=} \Box(\neg\mathsf{empty} \rightarrow p) \qquad p||q \stackrel{\text{def}}{=} p \wedge (q\,;\,\mathsf{true}) \vee q \wedge (p\,;\,\mathsf{true})$$
$$(p_1, \ldots, (p_i, \ldots, p_l)^{\circledast}, \ldots, p_m)\ \mathsf{prj}\ p \stackrel{\text{def}}{=} (p_1, \ldots, \varepsilon, \ldots, p_m)\ \mathsf{prj}\ p \vee (p_1, \ldots, (p_i, \ldots, p_l)^{\oplus}, \ldots, p_m)\ \mathsf{prj}\ p$$

A number of logic laws of PTL can be found in [6,7].

2.2 Modeling, Simulation and Verification Language

MSVL is an executable subset of PTL and used to model, simulate and verify concurrent systems. In addition, the variables within a program can refer to their previous values.

Syntax As an executable subset of PTL, MSVL consists of expressions and statements. Expressions can be regarded as PTL terms, and statements as PTL formulas. The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0\, op\, e_1 (op ::= + \mid - \mid * \mid / \mid mod)$$
$$b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows:

1. Assignment	$x = e$	2. P-I-Assignment	$x \Leftarrow e$	3. Conjunction	$p \wedge q$
4. Selection	$p \vee q$	5. State Frame	$\mathsf{lbf}(x)$	6. Termination	empty
7. Always	$\Box p$	8. Local variable	$\exists x : p$	9. Sequential	$p\,;\,q$
10. Next	$\bigcirc p$	11. Interval Frame	$\mathsf{frame}(x)$	12. Projection	$(p_1,\ldots,p_m)\ \mathsf{prj}\ q$

13. Parallel: $p \parallel q \stackrel{\text{def}}{=} p \wedge (q\,;\,\mathsf{true}) \vee q \wedge (p\,;\,\mathsf{true})$

14. Conditional: $\mathtt{if}\ b\ \mathtt{then}\ p\ \mathtt{else}\ q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$

15. While: $\mathtt{while}\ b\ \mathtt{do}\ p \stackrel{\text{def}}{=} (b \wedge p)^* \wedge \Box(\mathsf{empty} \rightarrow \neg b)$

16. Await: $\mathsf{await}(b) \stackrel{\text{def}}{=} (\mathsf{frame}(x_1) \wedge \ldots \wedge \mathsf{frame}(x_h)) \wedge \Box(\mathsf{empty} \leftrightarrow b)$ where $x_i \in V_b = \{x | x \text{ appears in } b\}$

where $p_1, \ldots, p_m$, p and q stand for programs of MSVL. The assignment $x = e$, $x \Leftarrow e$, empty, $\mathsf{lbf}(x)$, and $\mathsf{frame}(x)$ can be regarded as basic statements, and the others composite ones; $\mathsf{frame}(x)$ means that variable x always keeps its old value over an interval if no assignment to x is encountered.

Normal Form. Since PTL formulas are interpreted over intervals and MSVL is a subset of PTL, programs should be executed over a sequence of states which is a marked feature of MSVL programming. So execution of programs is implemented by reduction since a program can be reduced to two parts: present and remain. That fact can be illustrated by the normal form of programs given in [13].

Definition 1 (Normal Form of MSVL program). *An MSVL program q is in Normal Form if*

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{k} q_{ei} \wedge \mathsf{empty} \vee \bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj}$$

where $0 \leq k \leq 1$, $h \geq 0$, $k + h \geq 1$ and the following hold: q_{fj} is an internal program in which variables may refer to the previous states but not beyond the first state of the current interval over which the program is executed; each q_{ei} and q_{cj} is either true *or a state formula of the form $p_1 \wedge \ldots \wedge p_m$ $(m \geq 1)$ and each p_l $(1 \leq l \leq m)$ is either $(x = e)$ with $e \in D$, $x \in V$, or p_x or $\neg p_x$.*

Theorem 1. *For each MSVL program p there is a program p' in Normal Form satisfying $p \equiv p'$*

Theorem 1 tells us that for any MSVL program p there is a program q in the Normal Form such that $p \equiv q$. The proof of the theorem can be found in [6,7].

3 Timed Projection Temporal Logic

3.1 Syntax

TPTL is an extension of PTL. With TPTL, the time is discrete and linear and the time domain is $\mathbb{N}_\omega$. We only care about future time instants.

Π is still used as a countable set of *propositions*, and V is a countable set of typed static and dynamic *variables*. *term* e and *formula* p of PTL are defined as before, except for two newly added variables T and Ts. The *Time duration* $\mathfrak{D}$ and *TPTL formulas tp* are inductively defined as follows:

$$
\begin{array}{rcl}
e & ::= & v \mid \bigcirc e \mid \ominus e \mid \mathsf{beg}(e) \mid \mathsf{end}(e) \mid f(e_1, \ldots, e_m) \mid T \mid Ts \\
p & ::= & \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_m) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \bigcirc p \mid \\
 & & (p_1, \ldots, p_m)\ \mathsf{prj}\ p \mid (p_1, \ldots, (p_i, \ldots, p_l)^{\oplus}, \ldots, p_m)\ \mathsf{prj}\ p \\
\mathfrak{D} & ::= & (e_1, e_2) \\
tp & ::= & \mathfrak{D}p \mid \mathfrak{D}tp \mid tp_1 \wedge tp_2 \mid tp_1 \vee tp_2 \mid (tp_1, \ldots, tp_m)\ \mathsf{prj}\ tp \mid \\
 & & (tp_1, \ldots, (tp_i, \ldots, tp_l)^{\oplus}, \ldots, tp_m)\ \mathsf{prj}\ tp
\end{array}
$$

In a time duration $\mathfrak{D}$, e_1 and e_2 are the starting time and the ending time of the $\mathfrak{D}$ respectively. $\mathfrak{D}p$ and $\mathfrak{D}tp$ mean that a $\mathfrak{D}$ can be used to limit either PTL or TPTL formulas.

3.2 Semantics

With TPTL, we still use the quadruple $\mathfrak{I} = (\sigma, i, k, j)$ as an interpretation. An interpretation of the term e is the same as before. The newly-added terms can be interpreted as, like variables, $\mathfrak{I}[T] = s_k[T]$ and $\mathfrak{I}[Ts] = s_k[Ts]$ respectively. However, they have some special properties: Ts is a framed variable and always has a positive value while T increases monotonously as follows:

$$
\begin{cases}
\forall k(0 \leq k < |\sigma| \rightarrow s_{k+1}[T] = s_k[T] + s_k[Ts]) & \text{if the interval is finite} \\
\forall k(k \geq 0 \rightarrow s_{k+1}[T] = s_k[T] + s_k[Ts]) & \text{if the interval is infinite}
\end{cases}
$$

The satisfaction relation ($\models$) of $\mathfrak{D}p$ can inductively be defined as follows:

$\mathfrak{I} \models (t_1, t_2)p$ (resp. tp) if there exists l_1, l_2 such that $k \leq l_1 \preceq l_2 = j$ and $s_{l_1}[T] = \mathfrak{I}[t_1]$ and $s_{l_2}[T] = \mathfrak{I}[t_2]$ and $(\sigma, i, l_1, l_2) \models p$ (resp. tp)

Note that, in the interpretations, in order to distinguish between the time terms and ordinary terms, we use t_i to represent a time term. Consequently, (t_1, t_2) represents a time duration $\mathfrak{D}$. The other structures $tp_1 \wedge tp_2$, $tp_1 \vee tp_2$, $(tp_1, \ldots, tp_m)$ prj tp and $(tp_1, \ldots, (tp_i, \ldots, tp_l)^{\oplus}, \ldots, tp_m)$ prj tp have the same interpretations as before. Formula $(t_1, t_2)p$ (resp. $(t_1, t_2)tp$) means that the time duration (t_1, t_2) imposes time restrictions onto PTL or TPTL formula p (resp. tp). This also indicates that formula p (resp. tp) is interpreted from starting time $T = t_1$ to ending time $T = t_2$.

Theorem 2. *Let $\mathfrak{I}$ be an interpretation, and tp, tq TPTL formulas. If $\mathfrak{I} \models tq$ implies $\mathfrak{I} \models tp$, then $\mathfrak{I} \models tq \rightarrow tp$.*

Corollary 1. *Let σ be an interval (or a model). If $\sigma \models tq$ implies $\sigma \models tp$, then $\sigma \models tq \rightarrow tp$.*

In order to avoid an excessive number of parentheses, we use the precedence rules defined in [6]. Further, time duration can be regarded as a special unary operator and its priority is between $=$ and $\wedge$.

For convenience, in what follows, we use p, q or r (possibly with subscripts) to represent a TPTL formula or a PTL formula, w a state formula, lp a PTL formula, and tp or tq a TPTL formula when there is no explicit declaration. **p** and **q** stand for (possibly empty) sequences of formulas: $p_1, \ldots, p_m$ and $q_1, \ldots, q_n$.

3.3 Derived Formulas and Logic Laws

Formulas Derived from Duration Derivatives. Formulas derived from duration derivatives are constrained by time durations whose starting time or ending time is special terms, i.e., T, $\mathsf{end}(T)$ or $\bigcirc T$. At a state, these derived formulas

are equivalent to the elementary formulas when time related terms are replaced by the values of the special terms in the time durations. However, it is difficult to evaluate T in general with programming. If a formula's starting time is T, it is treated as starting at the current state; whereas, if a formulas's ending time is $\mathsf{end}(T)$, it terminates at the last state of the interval. Thus, a formula bounded with time duration $(T, \mathsf{end}(T))$ is not subject to any time constraints and it can be seen as a time independent formula. In this way, any PTL formula p can be treated as a special TPTL formula $(T, \mathsf{end}(T))p$. As a result, the set of PTL formulas can be viewed as a subset of TPTL formulas. The meanings of those derived formulas are given as follows.

1. $\mathfrak{I} \models (t, \mathsf{end}(T))p$ iff $(\sigma, i, l, j) \vDash p$ for $k \leq l \preceq j$ and $s_l[T] = \mathfrak{I}[t]$
2. $\mathfrak{I} \models (T, t)p$ iff $(\sigma, i, k, l) \vDash p$ for $k \leq l = j$ and $s_l[T] = \mathfrak{I}[t]$
3. $\mathfrak{I} \models (\bigcirc T, t)p$ iff $(\sigma, i, k+1, j) \vDash (T, t)p$
4. $\mathfrak{I} \models (T, \mathsf{end}(T))p$ iff $(\sigma, i, k, j) \vDash p$
5. $\mathfrak{I} \models (\bigcirc T, \mathsf{end}(T))p$ iff$(\sigma, i, k+1, j) \vDash (T, end(T))p$

The Abbreviations in TPTL. There are also some useful abbreviations below:

$lp \stackrel{\text{def}}{=} (T, \mathsf{end}(T))lp \quad \langle t_1, t_2\rangle lp \stackrel{\text{def}}{=} (t_1, t_2)\Diamond lp \quad [t_1, t_2]lp \stackrel{\text{def}}{=} (t_1, t_2)\Box lp$

$(\varepsilon)p \stackrel{\text{def}}{=} (T, T)p \quad p\,;\, q \stackrel{\text{def}}{=} (p, q)\ \mathsf{prj}\ \mathsf{empty} \quad p^{+} \stackrel{\text{def}}{=} (p^{\oplus})\ \mathsf{prj}\ \mathsf{empty}$

$p^{*} \stackrel{\text{def}}{=} (p^{\circledast})\ \mathsf{prj}\ \mathsf{empty} \quad p||q \stackrel{\text{def}}{=} p \wedge (q\,;\,\mathsf{true}) \vee q \wedge (p\,;\,\mathsf{true})$

$(\mathbf{p}, (p_i, \ldots, p_l)^{\circledast}, \mathbf{q})\ \mathsf{prj}\ p \stackrel{\text{def}}{=} (\mathbf{p}, \mathsf{empty}, \mathbf{q})\ \mathsf{prj}\ p \vee (\mathbf{p}, (p_i, \ldots, p_l)^{\oplus}, \mathbf{q})\ \mathsf{prj}\ p$

Theorem 3. *Let p, q be any formulas in PTL or TPTL, and (t_1, t_2) a time duration. Thus, $p \equiv q$ if and only if $(t_1, t_2)p \equiv (t_1, t_2)q$.*

Logic Laws

Theorem 4. *Let p, q be any formulas of TPTL, t, t_i be time terms, $i \in \mathbb{N}$. $T \leq t_i \leq t_{i+1}$. The following laws all hold.*

L1. $p \equiv (T, \mathsf{end}(T))p$
L2. $\mathsf{empty} \equiv (\varepsilon)\mathsf{empty}$
L3. $(t_1, t_3)(t_2, t_3)p \equiv (t_2, t_3)p$
L4. $(T, t) \bigcirc p \equiv (\bigcirc T, t)p$
L5. $(\varepsilon)w \equiv w \wedge \mathsf{empty}$
L6. $(\varepsilon)\mathsf{empty}\,;\, p \equiv p$
L7. $(t, t)(p \wedge \mathsf{empty}) \equiv (t, t)p$
L8. $(\bigcirc T, t)p \supset \mathsf{more}$
L9. $(t_1, t_2)p \wedge (t_1, t_2)q \equiv (t_1, t_2)(p \wedge q)$
L10. $(t_1, t_2)p \vee (t_1, t_2)q \equiv (t_1, t_2)(p \vee q)$
L11. $(t_1, t_2)p \wedge T = t_1 \equiv (T, t_2)p$
L12. $(T, t)w \equiv (\varepsilon)w \vee w \wedge (\bigcirc T, t)\mathsf{true}$
L13. $(t, \mathsf{end}(T))(p \wedge \mathsf{empty}) \equiv (t, t)(p \wedge \mathsf{empty})$
L14. $(t, \mathsf{end}(T))p \wedge (t_1, t_2)q \equiv (t, t_2)p \wedge (t_1, t_2)q$
L15. $(t_1, \mathsf{end}(T))(t_2, t_3)p \equiv (t_1, t_3)(t_2, t_3)p$
L16. $(\mathbf{p}, (\varepsilon)w, p, \mathbf{q})\ \mathsf{prj}\ q \equiv (\mathbf{p}, w \wedge p, \mathbf{q})\ \mathsf{prj}\ q$
L17. $(t_1, t_2) \bigcirc p \wedge T = t_1 \equiv T = t_1 \wedge (T + Ts, t_2)p$
L18. $(t_1, t_2)w \wedge T = t_1 \equiv T = t_1 \wedge ((\varepsilon)w \vee w \wedge (T + Ts, t_2)\mathsf{true})$
L19. $(t_1, t_3)p \wedge (t_2, t_3)q \equiv (t_1, t_3)((t_1, t_3)p \wedge (t_2, t_3)q)$
L20. $(t_1, t_2)p\,;\, (t_3, t_4)q \equiv (t_1, t_4)((t_1, t_2)p\,;\, (t_3, t_4)q)$
L21. $(p, \mathbf{p})\ \mathsf{prj}\ (t_1, t_2)q \wedge t_1 > T \equiv p\,;\, (\mathbf{p}\ \mathsf{prj}\ (t_1, t_2)q) \wedge t_1 > T$
L22. $(p, \mathbf{p})\ \mathsf{prj}\ (\bigcirc T, t)q \equiv p \wedge \mathsf{more}\,;\, \mathbf{p}\ \mathsf{prj}\ (T, t)q \vee p \wedge \mathsf{empty}\,;\, \mathbf{p}\ \mathsf{prj}\ (\bigcirc T, t)q$
L23. $(\bigcirc T, t_1)p\ \mathsf{prj}\ (\bigcirc T, t_2)q \equiv (\bigcirc T, t_1)p\,;\, (T, t_2)q$
L24. $((\bigcirc T, t_1)p_1, \mathbf{p})\ \mathsf{prj}\ (\bigcirc T, t_2)q \equiv (\bigcirc T, t_1)p_1\,;\, \mathbf{p}\ \mathsf{prj}\ (T, t_2)q$
L25. *Assuming $p_0 \equiv p_{m+1} \equiv \mathsf{empty}$:*

$$\mathbf{p}\ \mathsf{prj}\ (\bigcirc T, t)q \equiv \bigvee_{t=0}^{m-1} ((p_0 \wedge \ldots \wedge p_t) \wedge \mathsf{empty}\,;\, p_{t+1} \wedge \mathsf{more}\,;\, (p_{t+2}, \ldots, p_{m+1})\ \mathsf{prj}\ (T, t)q) \vee ((p_0 \wedge \ldots \wedge p_m)\,;\, (\bigcirc T, t)q)$$

Theorem 5. *The following logic laws in PTL which only contain the operators* ; , prj, $\wedge$, $\vee$, $+$ *and* $*$, *are valid in TPTL.*

L26.	$r\,;(p\vee q)\equiv(r\,;p)\vee(r\,;q)$	L33.	$p^*\equiv \mathsf{empty}\vee(p\,;p^*)\vee(p\wedge\Box\mathsf{more})$
L27.	$(p\vee q\,;r)\equiv(p\,;r)\vee(q\,;r)$	L34.	q prj empty $\equiv q$
L28.	$w\wedge(p\,;q)\equiv(w\wedge p)\,;q$	L35.	$\mathbf{p}$ prj empty $\equiv p_1\,;\,\ldots\,;\,p_m$
L29.	$p^+\equiv p\vee(p\,;p^+)$	L36.	$(\mathbf{p},\mathsf{empty},\mathbf{q})$ prj $q\equiv(\mathbf{p},\mathbf{q})$ prj q
L30.	$p^+\equiv p\vee((p\wedge\mathsf{more})\,;p^+)$	L37.	$\mathbf{p}$ prj $(p\vee q)\equiv(\mathbf{p}$ prj $p)\vee(\mathbf{p}$ prj $q)$
L31.	$p^+\wedge\mathsf{empty}\equiv p\wedge\mathsf{empty}$	L38.	$(w\wedge p,\mathbf{p})$ prj $q\equiv w\wedge((p,\mathbf{p})$ prj $q)$
L32.	p prj $q\equiv p\wedge q$	L39.	$\mathbf{p}$ prj $(w\wedge q)\equiv w\wedge(\mathbf{p}$ prj $q)$
L40.	$(\mathbf{p},p\vee r,\mathbf{q})$ prj $q\equiv((\mathbf{p},p,\mathbf{q})$ prj $q)\vee((\mathbf{p},r,\mathbf{q})$ prj $q)$		

4 Timed-MSVL

TMSVL is an executable subset of TPTL and also an extension of MSVL [10,11,8]. The arithmetic expression e and boolean expression b of TMSVL are the same as in MSVL. By the law $p\equiv(T,\mathsf{end}(T))p$ in logic, all of the MSVL statements can be used in TMSVL without changing their meaning. As a result, TMSVL can do whatever MSVL does.

4.1 Syntax and Semantics

In TMSVL, an expression e can be treated as a term and a statement p can be viewed as a formula in TPTL. There are ten elementary statements as follows:

1. MSVL statement	lp	2. Time limit	$(t_1,t_2)tp$
3. Conjunction	$tp_1\wedge tp_2$	4. Selection	$tp_1\vee tp_2$
5. Sequential	$tp_1\,;tp_2$	6. Point	$(\varepsilon)tp$
7. Projection	$(tp_1,\ldots,tp_m)$ prj tp	8. Conditional	`if` b `then` tp_1 `else` tp_2
9. While loop	`while` b `do` tp	10. Parallel	$tp_1\parallel tp_2$

Seven of them are constructs inherited from MSVL, which are conjunction($\wedge$), selection($\vee$), chop(;), projection(prj), conditional statement, while loop and parallel ($||$); MSVL statements can directly be used without changing their meaning; the other two statements indicate the statements under a time duration limitation or a point time constrains respectively.

Like MSVL, the execution of a TMSVL program P is to find an interval on which P can be satisfied. A program is executed either successfully or failed, depending on true or false being eventually reduced. There is difference between MSVL and TMSVL. In MSVL, a statement can be treated as a program. For a TMSVL program P, it consists of two parts: a clock generator $clock(e_T,e_{Ts})$ and TMSVL statement q. Thus, each TMSVL program has the form $clock(e_T,e_{Ts})\wedge q$. $clock(e_T,e_{Ts})$ is defined by the elementary statements below:

$$clock(e_T,e_{Ts})\stackrel{\text{def}}{=}T=e_T\wedge(\neg p_{Ts}\rightarrow Ts=e_{Ts})\wedge\mathsf{frame}(Ts)\wedge\mathsf{keep}(\bigcirc T=T+Ts)$$

which means that the value of T equals to the value of terms e_{Ts} at the current state. Since Ts is a framed variable, whenever $Ts=e$ is encountered, proposition p_{Ts} is set to true and Ts has the value e at the current state. If there is no assignment to Ts in the program, by using the minimal model semantics, p_{Ts} is forced to be false, and thus Ts has the value e_{Ts} at the current state. At the next state, the value of T is the sum of e_T and the current value of Ts.

Theorem 6. *Let p be a TMSVL statement and $P \equiv clock(e_T, e_{Ts}) \wedge p$ be a TMSVL program, and e'_{Ts} is an expression produced by p. Then we have*

$$P \equiv T = e_T \wedge Ts = e'_{Ts} \wedge p_{Ts} \wedge \odot clock(e_T + e'_{Ts}, e'_{Ts}) \wedge p \vee$$
$$T = e_T \wedge Ts = e_{Ts} \wedge \neg p_{Ts} \wedge \odot clock(e_T + e_{Ts}, e_{Ts}) \wedge p$$

4.2 Normal Form of Programs

Normal Form is of great importance to reduce programs over intervals. The Normal Form of TMSVL programs is a bit different from that of MSVL.

Fact 1. *Let p be a TMSVL statement. There exists a time duration (t_1, t_2) such that p can be translated to the form $(t_1, t_2)q$.*

Definition 2 (normal form of TMSVL programs). *A TMSVL program $P \equiv clock(e_T, e_{Ts}) \wedge (t_1, t_2)p$ is in Normal Form if*

$$P \equiv \Big(\bigvee_{i=1}^{k} (\varepsilon) p_{ei} \Big) \vee \Big(\bigvee_{j=1}^{h} p_{cj} \wedge \bigcirc clock(e_T + e'_{Ts}, e'_{Ts}) \wedge (t, t_2) p_{fj} \Big)$$

where $k + h \geq 1$, and the following holds: q_{fj} is a general TMSVL statement. q_{ei} and q_{cj} are state formulas of the form, $T = e_T \wedge Ts = e'_{Ts} \wedge x_1 = e_1 \wedge \ldots \wedge x_m = e_m \wedge p_T \wedge \dot{p_{Ts}} \wedge \dot{p_{x_1}} \wedge \ldots \wedge \dot{p_{x_m}}$. If $t_1 > T$, $t = t_1$, otherwise $t = T + Ts$ (i.e., $\bigcirc T$).

Theorem 7. *Let P be a TMSVL program. There is a program Q in Normal Form such that $P \equiv Q$.*

4.3 Operational Semantics of TMSVL

The Operational semantics of MSVL have been studied in [13]. Since we extend MSVL to TMSVL, the operational semantics of TMSVL should be reconsidered.

To execute a program P is really to find an interval to satisfy the program P. The execution of a program P consists of a series of reductions over a sequence of states, i.e., an interval. The reduction process is divided into two phases: one for state reduction and the other for interval reduction.

Configuration. Here we still use the concept of configuration in [13] to illustrate the operational semantics of TMSVL. A configuration regarding a program P is a quadruple $(P, \sigma_{i-1}, s_i, i)$, where P is a framed program, $\sigma_{i-1} = \langle s_0, \ldots, s_{i-1} \rangle (i > 0)$ a model which records information of all states, s_i the current state at which P is being executed and i a counter for counting the number of states in σ_{i-1}. When a program is terminating, it is reduced to true and the state is written as $\emptyset$. So the final configuration is $c_f = (\mathsf{true}, \sigma, \emptyset, |\sigma| + 1)$.

State Reduction. We first introduce semantic equivalence rules to normalize a program, and then specify the transition rules within a state to deal with the current assignment and catch the minimal models. All of these rules form a state reduction system. Because data structures, arithmetical expressions and boolean expressions of MSVL and TMSVL are all identical, so transition rules provided in [13] can be used to solve assignments and capture the minimal model with TMSVL [11]. In order to transform a program into its new normal form, we need semantic equivalence rules for the state reduction of TMSVL. These rules consists

of the existing rules given in [13], and new logic equivalence rules presented in section 3.3. These equivalence rules alter the form of P in a configuration $(P, \sigma_{i-1}, s_i, i)$ without changing the other three elements.

Example 1. *The state reduction for program* $P \stackrel{\text{def}}{=} clock(0,1) \wedge (1,2)\Box(x=1)$

$$\begin{aligned} P \equiv\ & clock(0,1) \wedge (1,2)\Box(x=1) \\ \equiv\ & T=0 \wedge \neg p_{Ts} \wedge Ts=1 \wedge \bigcirc clock(1,1) \wedge (1,2)\Box(x=1) \\ \equiv_m\ & p_T \wedge T=0 \wedge \neg p_{Ts} \wedge Ts=1 \wedge \bigcirc clock(1,1) \wedge (1,2)\Box(x=1) \end{aligned}$$

$P \equiv_m Q$ means P equals to Q under the minimal model [6]. From the reduction, we have $s_0 = \{p_T, Ts = 1, \neg p_{Ts}, T = 0\}$, s_0 represents the current state.

Interval Reduction. Once all of variables and propositions involved in the current state have been set, the remained subprogram is of the form $\bigcirc clock(e_T, e_{Ts}) \wedge (t_1, t_2)p$ or $(\varepsilon)\mathsf{empty}$. This means that the reduction of the program needs move to the next state or stop. We have two interval transition rules to describe the two situations:

$$\begin{array}{ll} \textsc{Tr1} & (\bigcirc clock(e_T, e_{Ts}) \wedge (t_1, t_2)p, \sigma_{i-1}, s_i, i) \\ & \rightarrow (clock(e_T, e_{Ts}) \wedge (t_1, t_2)p, \sigma_i, s_{i+1}, i+1) \qquad (\text{if } t_1 > T) \\ \textsc{Tr2} & (\bigcirc clock(e_T, e_{Ts}) \wedge (t_1, t_2)p, \sigma_{i-1}, s_i, i) \\ & \rightarrow (clock(e_T, e_{Ts}) \wedge (T, t_2)p, \sigma_i, s_{i+1}, i+1) \qquad (\text{if } t_1 \text{ is } \bigcirc T) \\ \textsc{Tr3} & ((\varepsilon)\mathsf{empty}, \sigma_{i-1}, s_i, i) \rightarrow (true, \sigma_i, \emptyset, i+1) \end{array}$$

Rule Tr1 and Tr2 are useful for dealing with $\bigcirc clock(e_T, e_{Ts}) \wedge (t_1, t_2)p$ while rule Tr3 is helpful for reducing $(\varepsilon)\mathsf{empty}$. The execution of $(\bigcirc clock(e_T, e_{Ts}) \wedge (t_1, t_2)p, \sigma_{i-1}, s_i, i)$ means that $clock(e_T, e_{Ts}) \wedge (t_1, t_2)p$ or $clock(e_T, e_{Ts}) \wedge (T, t_2)p$ requests to be executed at the next state s_{i+1}, according to Tr1 or Tr2 and current state s_i needs to be appended to model σ_{i-1}. So i, the number of states in σ_{i-1}, needs to be increased by one. The execution of $((\varepsilon)\mathsf{empty}, \sigma_{i-1}, s_i, i)$ is simple. State s_i is appended to σ_{i-1} and the final configuration $(true, \sigma_i, \emptyset, i+1)$ is reached. In this way, if a program is eventually reduced to true, then an interval which satisfies the program is obtained. Thus, $\sigma_i = \sigma_{i-1} \cdot \langle s_i \rangle$ is the model of the program.

Example 2. *Using the interval transition rules, we can continue with the reduction of the subprogram in Example 1.*

By the Tr1*, we know that at the second state the program needs to be reduced is $P_f^1 \equiv clock(1,1) \wedge (1,2)\Box(x=1)$. After the state reduction, we have*

$$P_f^1 \equiv_m \neg p_{Ts} \wedge p_T \wedge p_x \wedge Ts=1 \wedge T=1 \wedge x=1 \wedge \bigcirc clock(2,1) \wedge (2,2)\Box(x=1)$$

Thus, $s_1 = \{\neg p_{Ts}, Ts = 1, p_T, T = 1, p_x, x = 1\}$, and by the Tr1*, we obtain $P_f^2 \equiv clock(2,1) \wedge (2,2)\Box(x = 1)$*

After the state reduction, $P_f^2 \equiv_m \neg p_{Ts} \wedge Ts = 1 \wedge p_x \wedge p_T \wedge T = 2 \wedge x = 1 \wedge (\varepsilon)\mathsf{empty}$. Then, $s_2 = \{\neg p_{Ts}, Ts = 1, p_T, T = 2, p_x, x = 1\}$. By the Tr3*, the model of program P is obtained, $\sigma = \langle s_0, s_1, s_2 \rangle$.*

5 Applications

5.1 Description of Real-Time Systems

Safety, Liveness and Periodicity. Safety means something "bad" will never happen and liveness is meant by something "good" will eventually happen. MSVL statements can easily be used to specify these qualitative properties in terms of $\Box lp$ and $\Diamond lp$ while TMSVL statements are able to describe quantified safety and liveness by using time constraints $[t_1, t_2]lp$ and $\langle t_1, t_2\rangle lp$.

Periodicity is also an important property in real-time systems. The statement, `while` (`true`) `do` $(T, T+n)p$, can be used to describe some events (like p) occur periodically by a period of time n.

Interrupts. Interrupt handing is a frequently met problem in real-time programming. When an interrupt occurs, perhaps caused by a device requiring event, the running program is suspended and execution begins on the appropriate interrupt service routine. When the service routine finishes, the execution of the interrupted program resumes. This is just the kind of behavior conducted by the temporal projection statement. The execution of the interrupt service occurs between two consecutive states of the interrupted program.

In the statement $(p_1, \cdots, p_m)$ prj q, $p_1; \cdots; p_m$ and q are started at the same time but may terminate at different time points. A statement $(p^{\circledast})$ prj q means that at each beginning of the execution of p, a reduction of q is executed, and q may terminate at any state, but p is still executed repeatedly. However, the statement $(p^{\circledast},\ r \wedge \varepsilon)$ prj $(q\,;\, r \wedge \varepsilon) \wedge \mathsf{halt}(r)$ prevents the repetitive execution of p when q terminates; r is a proposition used to mark the end of q.

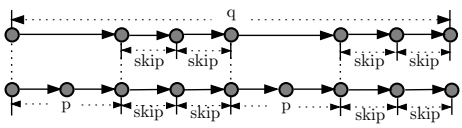


Fig. 1. Construction of the formula q `when` b `do` p

For convenience, we write q `when` b `do` p to mean that whenever the boolean expression b is `true`, the execution of q is interrupted by p.

$$q \texttt{ when } b \texttt{ do } p \stackrel{\text{def}}{=} ((\texttt{if } b \texttt{ then } p \texttt{ else skip})^{\circledast},\ r \wedge \varepsilon) \text{ prj } (q\,;\, r \wedge \varepsilon) \wedge \mathsf{halt}(r)$$

Interrupt handling is easily described by q `when` b `do` p. How it works is illustrated in figure 1.

Timeout. Sometimes it is necessary to limit the time spent waiting for a particular condition to become true or to terminate the program when time is out ignoring the status of its execution. Time out is a problem often occurred in real-time programming. If $clock(e_1, e_2) \wedge p \equiv c \wedge p$ is a deterministic program, a timeout time constraint is defined as follows:

$$c \wedge (t_1 \ @\ t_m)p \stackrel{\text{def}}{=} c \wedge (t_1, t_m)\ p_c^1 \wedge (t_2, t_m)p_c^2 \wedge \ldots \wedge (t_m, t_m)(p_e^m \vee p_c^m)$$

where $t_1, \ldots, t_m$ are the time values of m consecutive states respectively, p_c^i represents a state formula after the state reduction when $T = t_i$ $(1 \leq i \leq m)$; p_e^m

represents a terminal state formula when $T = t_m$. They can be defined inductively below.

Let P be a deterministic TMSVL program, and according to definition 2 we know that $clock(t_1, e_{Ts1}) \wedge p$ can be transformed to Normal Form

$$clock(t_1, e_{Ts1}) \wedge p \equiv c^0 \wedge p_f^0 \equiv (\varepsilon)p_e \vee p_c \wedge \bigcirc clock(t_2, e_{Ts2}) \wedge (t_2, t)p_f$$

Then,$p_e^1 \equiv p_e$, $p_c^1 \equiv p_c$, $p_f^1 \equiv (t_2, t)p_f$, $c^1 \equiv clock(t_2, e_{Ts2})$.

Suppose $c^n \wedge p_f^n \equiv (\varepsilon)q_e \vee q_c \wedge \bigcirc clock(t_n, e_{Tsn}) \wedge (t_n, t)q_f$, $0 \leq n < m$. Then we have $p_e^{n+1} \equiv q_e$, $p_c^{n+1} \equiv q_c$, $p_f^{n+1} \equiv (t_n, t)q_f$,$c^{n+1} \equiv clock(t_n, e_{Tsn})$.

A timeout related to a deterministic program is easily defined using $(t_1@t_m)p$, which means that statement p would be terminated, when $T = t_m$.

Theorem 8. *$clock(e_1, e_2) \wedge p \equiv (\varepsilon)p_e^1 \vee p_c^1 \wedge \bigcirc clock(e_1', e_2') \wedge p_f^1$, and t_i, $t_{i+1}(1 \leq i \leq m-1)$, p_e^1, p_c^1 and p_f^1 are defined as the above. The following formulas are valid.*

$$clock(e_1, e_2) \wedge (t_1 \ @ \ t_1)p \equiv clock(e_1, e_2) \wedge (t_1, t_1)(p_e^1 \vee p_c^1)$$
$$clock(e_1, e_2) \wedge (t_i \ @ \ t_m)p \equiv clock(e_1, e_2) \wedge (t_i, t_m)(p_c^1 \wedge (t_{i+1} \ @ \ t_m)p_f^1)$$

A nondeterministic program can be written to the form $clock(e_1, e_2) \wedge p \equiv \bigvee_{i=1}^{k} clock(e_T, e_{Ts}) \wedge q_i$, $clock(e_T, e_{Ts}) \wedge q_i$ is a deterministic program. Thus, we can use the definition of timeout constraint on a deterministic program to define timeout on a nondeterministic program as follows:

$$clock(e_1, e_2) \wedge (t_1@t_m)p \equiv \bigvee_{i=1}^{k} clock(e_T, e_{Ts}) \wedge (t_1@t_m)q_i$$

Delay. With TMSVL, the time duration constrains the formula in a rigorous way, and time delay is usually allowed. We assume that the finishing time of a statement p is after t_1 and can be delayed until after t_m. It means statement p starting from now on and ending between $T + t_1$ and $T + t_m$. Since the time increment Ts might be changed by a programmer, it is impossible to predicate the time values at states between two time points. Suppose Ts is not changed during the execution of p; thus we can solve the problem. Then we define a delay time constraint as follows:

$\{t_1, t_m\}p \stackrel{\text{def}}{=} t = Ts \wedge \mathsf{keep}(\bigcirc Ts = t) \wedge \big((T, T+t_1)p \vee \ldots \vee (T, T+t_i)p \vee \ldots \vee (T, T+t_m)p\big)$
where $t_i = t_1 + (i-1) * Ts$ and $1 \leq i \leq m$.

Timeout after Delay. We can also define a new time constraint called *the timeout after the time delay*:

$\{t_1, t_m\}p \stackrel{\text{def}}{=} t = Ts \wedge \mathsf{keep}(\bigcirc Ts = t) \wedge \big((T, T+t_1)p \vee \ldots \vee (T, T+t_i)p \vee \ldots \vee (T@T+t_m)p\big)$
where $t_i = t_i + (i-1) * Ts$ and $1 \leq i \leq m$. The formula $\{t_1 \ @ \ t_m\}p$ means that statement p is either terminated within the time duration $(T + t_1, T + t_m)$ or forced terminated immediately after t_m time units.

5.2 A Video-On-Demand System

An interactive video-on-demand (VOD) system allows users to access video services, such as movies, electronic encyclopedia, interactive games, and educational

videos from video servers on a broadband network. To model, simulate and verify these types of complicated systems by means of TMVSL, we present a simplified VOD system for specification and verification using the technique proposed in this paper in the following.

Requirement

- A video application provides movies to users.
- On users selecting a movie, the application turns to the loading status and sends a request to the server for the resource, and then waits for a response.
- Once the server receives a request, the selected movie would be sought out and a connection with the application could be set up within 2 to 3 seconds.
- If the application receives the answer in 3 seconds, the movie starts otherwise the application is timeout.
- Operation of pause is admitted during the playing.
- The application quits automatically when a movie has been played to the end.

Modeling. Before modeling, time variables and some other variables need to be initialized in terms of a program P_0:

$$P_0 \stackrel{\text{def}}{=} clock(0,1) \wedge \mathsf{frame}(req1, req2, movID, remtime, appstate, serstate, MovTime[5], conOK, stop) \wedge req1 = 0 \wedge req2 = 0 \wedge movID = -1 \wedge remtime = 0 \wedge MovTime[5] = (3,4,4,6,2)$$

where $req1$ and $req2$ are used to send requests to the application and the server respectively, movID to send the movie ID; $remtime$ is the remaining time of the playing movie; $appstate$ and $serstate$ represent states of the application and server respectively; array $MovTime[5]$ records the movies information in the server; $conOK$ means if the connection between application and server is OK; $stop$ is used to send a signal of pause; $\mathsf{frame}(X)$ is used to declare that the variables in X are framed.

The application can be modeled by a TMSVL program P_1 as shown below:

$$\begin{array}{l} P_1 \stackrel{\text{def}}{=} \mathtt{while}(\mathsf{true}) \\ \quad \mathtt{do}\{\mathsf{await}(req1 = 1)\,;\, req2 := 1\,; \\ \quad\quad \{0@3\}\{\ \mathtt{while}(!conOK)\ \mathtt{do}\ appstate := \text{`}loading\text{'}\ \}\,; \\ \quad\quad \mathtt{if}(conOK) \\ \quad\quad \mathtt{then}\{appstate := \text{`}playing\text{'}\,; \\ \quad\quad\quad \{\mathtt{while}(remtime > 0) \\ \quad\quad\quad\quad \mathtt{do}\{remtime := remtime - Ts \wedge appstate := \text{`}playing\text{'}\} \\ \quad\quad\quad \}\mathtt{when}(stop = 1)\ \mathtt{do} \\ \quad\quad\quad \{\ appstate := \text{`}pause\text{'}\,; \\ \quad\quad\quad\quad \mathsf{await}(stop = 0)\,; \\ \quad\quad\quad\quad appstate := \text{`}playing\text{'}\} \\ \quad\quad\quad \}\quad\}\,; \\ \quad\quad\quad appstate := \text{`}ready\text{'} \wedge req1 := 0 \wedge stop := 0 \wedge movID := -1\,;\, \mathsf{empty}\} \end{array}$$

The server can be modeled by a TMSVL program P_2 presented below:

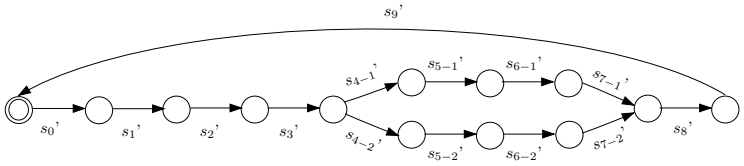


Fig. 2. NFG of program P: A node represents a state reduction, and an arc represents an interval transition. $s_i'(s_i' = s_i - \{T = e_T\})$ on the arrows represents the states before transition and an infinite path means a model of the system P

$$P_2 \stackrel{\text{def}}{=} \texttt{while}(\texttt{true})$$
$$\texttt{do}\{serstate := \text{`}waiting\text{'} \wedge conOK := 0 \wedge req2 := 0\,; \quad \texttt{await}(req2 = 1)\,;$$
$$\texttt{if } (movID >= 0 \wedge movID <= 4)$$
$$\texttt{then } \{2,3\}\{\ \texttt{next } serstate = \text{`}connecting\text{'} \wedge$$
$$\texttt{next } remtime = MovTime[movID]\,;\ conOK := 1\ \}$$
$$\texttt{else empty}\}$$

The operations of a user can be described by a TMSVL program P_3:

$$P_3 \stackrel{\text{def}}{=} \texttt{while}(\texttt{true})\ \texttt{do}\ \{\ (T, T+10)\{\ \texttt{next}\ req1 = 1 \wedge \texttt{next}\ movID = 4\ \}\ \}$$

As a result, the whole system can be modeled by a TMSVL program $P \equiv P_0 \wedge (P_1 \parallel P_2 \parallel P_3)$.

Verification. Suppose the property to be verified is defined by a TPTL formula φ. To verify whether P satisfies φ amounts to proving program P implies φ. In other words, we have to prove $\models P \rightarrow \varphi$. To do so, firstly, the program P can be reduced by its operational semantics step by step; then we can capture the set M of all models conducted by P from its Normal Form Graph (NFG) [11].

Here, we omit the tedious process of the reduction and show the result in table 1 which lists the values of the related variables in the beginning at several states of the models. During the reduction, there are loops in the NFG and the state s_{10k+i} is the same as s_i $(0 \leq k,\ 0 \leq i \leq 9)$ except for the time value. Thus, in table 1, it is not necessary for us to present all states in the NFG.

Table 1. variable's value at each state in model σ

T	state	req1	req2	serstate	appstate	conOK	movID	remtime	···
0	s_0	0	0	waiting	ready	0	-1	0	···
1	s_1	1	0	waiting	ready	0	4	0	···
2	s_2	1	1	waiting	loading	0	4	0	···
3	s_3	1	1	connecting	loading	0	4	2	···
4	s_{4-1}	1	1	connecting	loading	1	4	2	···
4	s_{4-2}	1	1	connecting	loading	0	4	2	···
5	s_{5-1}	1	0	waiting	playing	0	4	2	···
5	s_{5-2}	1	1	connecting	loading	1	4	2	···
6	s_{6-1}	1	0	waiting	playing	0	4	1	···
6	s_{6-2}	1	0	waiting	playing	0	4	2	···
7	s_{7-1}	0	0	waiting	ready	0	-1	0	···
7	s_{7-2}	1	0	waiting	playing	0	4	1	···
8	s_8	0	0	waiting	ready	0	-1	0	···
9	s_9	0	0	waiting	ready	0	-1	0	···
10	s_{10}	0	0	waiting	ready	0	-1	0	···
11	s_{11}	1	0	waiting	ready	0	4	0	···
···									

We can verify the properties like φ_1 and φ_2 below:

$$\varphi_1 \stackrel{\text{def}}{=} \Box(serstate = \text{`}connecting\text{'} \rightarrow appstate = \text{`}loading\text{'})$$
$$\varphi_2 \stackrel{\text{def}}{=} (\langle T, T+10\rangle(appstate = \text{`}playing\text{'} \wedge movID = 4))^+$$

Note that, as a matter of fact, property φ_1 is a safety property which can also be specified by LTL and CTL while property φ_2 is a full regular property (here a Kleen closure property) which LTL and CTL are both failed to define it.

6 Related Work

In the past three decades, many researchers have explored temporal representation of real-time systems. There are many real-time logics including LTLC[15], MTL[4], RTCTL[16], TPTL[3], TCTL[17], TRIO[20] and so on. However, none of these logics can be used as a programming language except TRIO. Their primary functions are specification and verification of real-time systems.

The model of timed automata for real-time systems was first proposed by Alur and Dill[14]. A timed automaton is a finite-state automaton equipped with a finite set of clocks which can hold non-negative real values. It is structured as a directed graph whose nodes are modes(control locations) and whose arcs are transitions. In practice, a real-time system is usually described as a set of process timed automata, each representing the behavior of an autonomous process. There are also some verification tools for timed systems based on timed automata, like UPPAAL which can be used to verify the property specified by CTL. However, timed automata are not executable, so they cannot be used as a programming language and are not capable of doing simulation directly.

Temporal logic formulas can also be used to describe system behaviors. In such frameworks, the system descriptions, including the models for systems and environments, and the properties are all in the same language, which facilitates a proof system established and makes the verification easier. The existing temporal logic programming languages like XYZ/E, TLA and TRIO can be used to model a real-time system.

XYZ/E [21] is an executable temporal logic language put forward by ZS. Tang. XYZ/RE extended the temporal operator \$◯, \$◇, \$□, \$$U$, \$$W$ in XYZ/E in order to make them capable to express the real time lower limit (l) and upper limit (u). Thereby, it gets the corresponded form of the real time operator: \$◯$\{l, u\}$, \$◇$\{l, u\}$, \$□$\{l, u\}$, \$$U\{l, u\}$, \$$W\{l, u\}$. XYZ/RE adopts time constraints in an intuitive way which makes it easy to understand. However, because its underlying logic is un-timed LTL, the sequential statement and the loop statement suffer from high expression complexity. Meanwhile, users have to repeatedly reset the time limitation of a statement in order to meet the consistency requirement.

Temporal Logic of Actions (TLA)[18] does not have a quantitative treating of time. In[19] Abadi and Lamport show how to introduce a distinguished state variable *now* with a continuous domain to represent the current time. There is no meaning for the notion of the 'next state'. In stead, prime (') is used to represent values of variables in a new state. TLA formulas can be expressed in terms of

familiar mathematical operators (such as $\wedge$) plus three new ones: $'$ (prime), $\Box$, and $\exists$. This makes TLA satisfy the fairness and more concise. However, this also makes it difficult to describe the duration of an action and the interval between two actions.

TRIO[20] is an extension of FOL (First Order Logic), and TRIO introduces a quantitative notion of time by adopting a single basic modal operator, called $Dist$. The simplest formula $Dist(p, d)$ means that proposition p holds at a time instant exactly d time unites from the current one; notice that this formula may refer to the future, if $d>0$, or to the past, if $d<0$, or even to the present time if $d=0$. Since the adoption of time points instead of intervals, it leads to increasing complexity. For instance, as for basic formula $(t1, t2)\Diamond p$ in TPTL, in TRIO the formula $Dist(\forall t'(0<t'<t_2-t_1 \rightarrow Dist(p, t')), t_1)$ represents the same meaning.

7 Conclusion

In this paper, we have extended the temporal logic PTL with time constraints and proposed a real-time programming language TMSVL used for modeling, simulation and verification of real-time systems. Further, the operational semantics of TMSVL has also been formalized. Based on the semantics, a prototype of TMSVL has been developed recently. In the future, we will further formalize an axiom system of TMSVL so that theorem proving approach can be conducted to verify real-time systems. Moreover, we plan to develop a model checker for TMSVL in order to verify real-time systems by means of the model checking approach. Finally, as case studies, we will model, simulate and verify practical examples to exam our approach proposed in this paper.

References

1. Melliar-Smith, P.M.: Extending interval logic to real time systems. In: Proceedings of the Conference on Temporal Logci Specification, UK, pp. 224–242. Springer (April 1987)
2. Razouk, R., Gorlick, M.: Real-time interval logic for reasoning about executions of real-time programs. SIGSOFT Softw. Eng. Notes 14(8), 10–19 (1989)
3. Alur, R., Henzinger, T.A.: A really temporal logic. In: Proceedings of the 30th IEEE Conference on Foundations of Computer Science. IEEE Computer Society Press, Los Alamitos (1989)
4. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
5. Mattonlini, R., Nesi, P.: An Interval Logic for Real-Time System Specification. IEEE Trans. Softw. Eng. 27, 208–227 (2001)
6. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD Thesis, University of Newcastle upon Tyne (1996)
7. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)
8. Duan, Z., Koutny, M.: A Framed Temporal Logic Programming Language. Journal of Computer Science and Technology 19, 341–351 (2004)

9. Duan, Z., Koutny, M., Holt, C.: Projection in Temporal Logic Programming. In: Pfenning, F. (ed.) LPAR 1994. LNCS (LNAI), vol. 822, pp. 333–344. Springer, Heidelberg (1994)
10. Duan, Z., Yang, X., Koutny, M.: Semantics of Framed Temporal Logic Programs. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 356–370. Springer, Heidelberg (2005)
11. Duan, Z., Yang, X., Koutny, M.: Framed Temporal Logic Programming. Science of Computer Programming 70(1), 31–61 (2008)
12. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press (1986)
13. Yang, X., Duan, Z.: Operational semantics of Framed Tempura. J. Log. Algebr. Program. 78(1), 22–51 (2008)
14. Alur, R., Dill, D.: Automata for Modeling Real-Time Systems. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 322–335. Springer, Heidelberg (1990)
15. Kwon, Y., Agha, G.: LTLC: Linear Temporal Logic for Control. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 316–329. Springer, Heidelberg (2008)
16. Emerson, E.A., Mok, A.K., Sistla, A.P., Srinivasan, J.: Quantitative temporal reasoning. Real-Time Systems 4, 330–352 (1992)
17. Alur, R., Courcoubetis, C., Dill, D.L.: Model checking for real-time systems. In: Proc. IEEE Fifth Symp. Logic in Computer Science, pp. 414–425 (1990)
18. Lamport, L.: The temporal logic of action. ACM Transations on Programming Languages and Ssystems 16(3), 872–923 (1994)
19. Abadi, M., Lamport, L.: An old-fashioned recipe for real time. ACM Transactions on Programming Languages and Systems 16(5), 1543–1571 (1994)
20. Ghezzi, C., Mandrioli, D., Morzenti, A.: TRIO: A logic language for executable specifications of real-time systems. The Journal of Systems and Software 12(2), 107–123 (1990)
21. Tang, Z.S.: Temporal Logic Program Designing and Engineering, vol. 1. Science Press, Beijing (1999)

Stepwise Satisfiability Checking Procedure for Reactive System Specifications by Tableau Method and Proof System

Yoshinori Neya and Noriaki Yoshiura

Department of Information and Computer Science, Saitama University
255, Shimo-ookubo Urawa-ku, Saitama City, Japan

Abstract. Open reactive systems are systems that ideally never terminate and are intended to maintain some interaction with their environment. Temporal logic is one of the methods for formal specification description of open reactive systems. For an open reactive system specification, we do not always obtain a program satisfying it because the open reactive system program must satisfy the specification no matter how the environment of the open reactive system behaves. This problem is known as realizability and the complexity of realizability check is double or triple exponential time of the length of specification formula and realizability checking of specifications is impractical. This paper implements stepwise satisfiability checking procedure with tableau method and proof system. Stepwise satisfiability is one of the necessary conditions of realizability of reactive system specifications. The implemented procedure uses proof system that is introduced in this paper. This proof system can accelerate the decision procedure, but since it is imcomplete it cannot itself decide the realizability property of specifications. The experiment of this paper shows that the implemented procedure can decide the realizability property of several specifications.

Keywords: Reactive System, Temporal Logic, Realizability.

1 Introduction

Open reactive systems, such as operating systems or elevator control systems, are systems that ideally never terminate and are intended to maintain some interaction with their environment and provide services. Open reactive system behavior depends on this interaction [4]. In reactive systems, events are divided into output events and input events. Output events are controlled or generated by reactive systems and input events are generated by the environment such as users of reactive systems. Input events cannot be controlled by the system [5].

Temporal logic is one of the methods of describing open reactive system specifications. One of the advantages of temporal logic is to prove several properties of the specifications. Given an open reactive system specification, which is a

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 283–298, 2012.

temporal logic formula, satisfiability of the formula only guarantees possibility of synthesizing a reactive system that satisfies a reactive system specification for some environment behavior. However, it is necessary to synthesize a system that satisfies the specification for all environment behaviors; suppose that there are finite sets I and O of input and output events. An open reactive system program is a function $f : (2^I)^* \to 2^O$ that maps finite sequences of input event sets into an output event set. Since an environment generates infinite sequences of input event sets, the function f generates infinite sequences of $2^{I \cup O}$ during interaction between an open reactive system and an environment.

The function f is a synthesized program satisfying a specification ϕ if and only if ϕ is always true for all infinite sequences generated by the function f and the environment behavior. The realizability problem is to determine, given a specification ϕ which is a temporal logic formula, whether there exists a program $f : (2^I)^* \to 2^O$ such that all of infinite sequences generated by f satisfies ϕ for any infinite sequence of input events made by the environment.

There are two kinds of temporal logic, linear and branching temporal logic. For the realizability problem of branching time temporal logic, there are several studies [1,5,8]. In these studies, computation tree and automata theory are used and especially, the realizability decision problems for CTL and CTL* are 2EXPTIME-complete and 3EXPTIME-complete [5]. There are also several studies for linear time temporal logic [7]. In a few years, the realizability for Metric Temporal Logic have been studied [2,6]. Metric Temporal Logic is a linear time timed temporal logic which extends linear time temporal logic with timing constraints on Until operator. Realizability of a specification described by Metric Temporal Logic is undecidable and that of Safety-MTL, which is a subset of Metric Temporal Logic, is undecidable if a certain kind of restriction is not added.

In previous studies on realizability of open reactive system specification described by several kinds of temporal logic, a main focus is on decidability of realizability and not on speed of deciding realizability. Automata theory has been used in almost all studies to give decision procedure of realizability. In order to decide the properties of realizability, using automata theory is effective, however, it is insufficient to decide realizability of a specification fast. There are several researches that are related with implementation of realizability decision [14]. Tableau method is important for realizability decision and there are several researches that are related with tableau fast construction [15].

The aim of this paper is to develop fast realizability decision procedure, but complexity of realizability decision shows that implementation of decision procedures is impractical. Thus, this paper implements a decision procedure of stepwise satisfiability, which is one of necessity conditions of realizability. Many of actual reactive system specifications are stepwisely satisfiable but not realizable [12]. This paper introduces a proof system for realizability to develop a fast decision procedure. This proof system can decide unrealizability of several specifications. The decision procedure that is implemented in this paper consists

of tableau method and proof system. Thus, the decision procedure can decide realizability of several specifications fast.

This paper is organized as follows. Section 2 explains open reactive system and section 3 explains temporal logic that is used for describing specifications. Section 4 shows tableau based decision procedure and section 5 introduces a proof system for realizability of reactive system specifications. Section 6 explains implementation of the stepwise satisfiability decision procedure and section 7 shows the results of the experiment. Section 8 concludes this paper.

2 Open Reactive System

This section provides a formal definition of an open reactive system, based on references [8]. Let A be a finite set. A^+ and A^ω denote the set of *finite sequences* and the set of *infinite sequences* over A respectively. $A^\dagger$ denotes $A^+ \cup A^\omega$. Sequences in $A^\dagger$ are indicated by $\hat{a}, \hat{b}, \cdots$, sequences in A^+ by $\bar{a}, \bar{b}, \cdots$ and sequences in A^ω by $\tilde{a}, \tilde{b}, \cdots$. $|\hat{a}|$ denotes the *length* of $\hat{a}$ and $\hat{a}[i]$ denotes the *i-th element* of $\hat{a}$. If B is a set whose elements are also sets, '$\sqcup$' is defined over $B^\dagger \times B^\dagger$ by

$$\hat{a} \sqcup \hat{b} = \hat{a}[0] \cup \hat{b}[0],\ \hat{a}[1] \cup \hat{b}[1],\ \hat{a}[2] \cup \hat{b}[2], \cdots.$$

Definition 1. *An open reactive system RS is a triple* $RS = \langle X, Y, r \rangle$*, where*

- X *is a finite set of input events produced by the environment.*
- Y $(X \cap Y = \emptyset)$ *is a finite set of output events produced by the system itself.*
- $r : (2^X)^+ \to 2^Y$ *is a reaction function.*

A subset of X is called an *input set* and a sequence of input sets is called an *input sequence*. Similarly, a subset of Y is called an *output set* and a sequence of output sets is called an *output sequence*. In this paper, a reaction function corresponds to an open reactive system program.

Definition 2. *Let* $RS = \langle X, Y, r \rangle$ *be an open reactive system and* $\hat{a} = a_0, a_1, \cdots \in (2^X)^\dagger$ *be an input sequence. The behavior of RS for* $\hat{a}$*, denoted* $behave_{RS}(\hat{a})$*, is the following sequence:*

$$behave_{RS}(\hat{a}) = \langle a_0, b_0 \rangle, \langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle, \ldots,$$

where for each i $(0 \leq i < |\hat{a}|)$, $b_i = r(a_0, \ldots, a_i) \in 2^Y$.

3 Specification

This paper uses propositional linear-time temporal logic (PLTL) as a specification language for open reactive systems.

3.1 Syntax

A PLTL *formula* is defined as follows:

- An atomic proposition $p \in \mathcal{P}$ is a formula.
- If f_1 and f_2 are formulas, $f_1 \wedge f_2$, $\neg f_1$ $f_1 \mathcal{W} f_2$ are formulas.

This paper uses "Weak until operator" ($\mathcal{W}$). This paper also uses abbreviation; $\Box f_1 \equiv f_1 \mathcal{W}(f_2 \wedge \neg f_2)$, $A\overline{\mathcal{W}}B \equiv \neg(\neg A \mathcal{W} \neg B)$ and $\vee$, $\rightarrow$, $\leftrightarrow$ and $\Diamond$ are the usual abbreviations.

3.2 Semantics

This subsection defines the semantics of PLTL formula on the behaviors. Let P be a set of events and $\mathcal{P}$ be a set of atomic propositions corresponding to each element of P. $\langle \sigma, i \rangle \models f$ denotes that a formula f over $\mathcal{P}$ holds at the i-th state of a behavior $\sigma \in (2^P)^\omega$. $\langle \sigma, i \rangle \models f$ is recursively defined as follows.

- $\langle \sigma, i \rangle \models p$ **iff** $p' \in \sigma[i]$ ($p \in \mathcal{P}$ is an atomic proposition corresponding to $p' \in P$)
- $\langle \sigma, i \rangle \models \neg f$ **iff** $\langle \sigma, i \rangle \not\models f$.
- $\langle \sigma, i \rangle \models f_1 \wedge f_2$ **iff** $\langle \sigma, i \rangle \models f_1$ and $\langle \sigma, i \rangle \models f_2$.
- $\langle \sigma, i \rangle \models f_1 \mathcal{W} f_2$ **iff** $(\forall j \geq 0)\ \langle \sigma, i+j \rangle \models f_1$ or $(\exists j \geq 0)\ (\ \langle \sigma, i+j \rangle \models f_2$ and $\forall k (0 \leq k < j)\ \langle \sigma, i+k \rangle \models f_1)$.

σ is a model of f if and only if $\langle \sigma, 0 \rangle \models f$. We write $\sigma \models f$ if $\langle \sigma, 0 \rangle \models f$. A formula f is satisfiable if and only there is a model of f. For example, $behave_{RS}(\tilde{a}) \models \varphi$ means that a behavior made by the reactive system RS receiving the input sequence $\tilde{a}$ is a model of φ.

3.3 Specification

A *PLTL-specification* for an open reactive system is a triple $Spec = \langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$, where

- $\mathcal{X}$ is a set of *input propositions* that are atomic propositions corresponding to the input events of the intended open reactive system, i.e. the truth value of an input proposition represents the occurrence of the corresponding input event.
- $\mathcal{Y}$ is a set of *output propositions* that are atomic propositions corresponding to the output events of the intended open reactive system, i.e. the truth value of an output proposition represents the occurrence of the corresponding output event.
- φ is a formula in which all the atomic propositions are elements of $\mathcal{X} \cup \mathcal{Y}$.

This paper writes $Spec = \langle \mathcal{X}, \mathcal{Y}, \varphi \rangle$ just as φ if there is no confusion. Finally, the following defines the realizability of open reactive system specifications [8].

Definition 3. *An open reactive system RS is an* **implementation** *of a specification φ if for every input sequence $\tilde{a}$, $behave_{RS}(\tilde{a}) \models \varphi$. A specification is* **realizable** *if it has an implementation.*

3.4 Stepwise Satisfiability

Reactive system specifications can be divided into several classes. This subsection explains the class of *stepwise satisfiability* specifications. This property is introduced in references [9]. Throughout this subsection $\tilde{a}$, $\bar{a}$ and $\tilde{b}$ denote an infinite input sequence, a finite input sequence and an infinite output sequence respectively. Also, r denotes a reaction function. For simplicity the interpretation representing a behavior is denoted by the behavior itself. Stepwise satisfiability is defined as follows:

Definition 4. *A reactive system RS preserves satisfiability of φ if and only if $\forall\bar{a}\exists\tilde{a}\exists\tilde{b}(behave_{RS}(\bar{a})(\tilde{a} \sqcup \tilde{b}) \models \varphi)$. φ is stepwisely satisfiable if and only if there is a reactive system which preserves the satisfiability of φ.*

If φ is stepwisely satisfiable, there is a reactive system $RS = \langle X, Y, r\rangle$ which preserves the satisfiability of φ. For any input event sequence, the reactive system RS can behave with keeping a possibility that φ is satisfied even though RS actually does not satisfy φ. The following shows two examples of specification to explain stepwise satisfiability.

EXAMPLE 1. *Satisfiable but not stepwisely satisfiable specifications.*

1. $\boldsymbol{req_1} \rightarrow \Diamond res$.
2. $\boldsymbol{req_2} \rightarrow \Box\neg res$.

$\boldsymbol{req_1}$ and $\boldsymbol{req_2}$ are input propositions and res is an output proposition. This example shows that a specification is not stepwisely satisfiable if it could require conflicting responses at the same time.

EXAMPLE 2. *Stepwisely satisfiable but not realizable specification.*

This example shows a part of a simple lift specification. In the specification below, an output proposition $Move$ is intended to show when the lift can move. $Open$ means that the lift door is open, and $Floor_i$ means that the lift is on the i-th floor. An input proposition $\boldsymbol{B_{open}}$ represents the request "open the door" and $\boldsymbol{B_i}$ represents the request "come or go to the i-th floor."

1. $\Box(\neg Move \wedge Floor_i \rightarrow Floor_i \,\mathcal{W}\, Move)$
   (if $Move$ is not true when the lift is at the i-th floor, stay there until $Move$ holds)
2. $\Box(Open \rightarrow \neg Move)$
   (if the door is open, do not move)
3. $\Box\big(\neg\boldsymbol{B_{open}}\wedge(\neg Move\mathcal{W}\boldsymbol{B_{open}}) \rightarrow (\neg Open\,\mathcal{W}\,(\boldsymbol{B_{open}}\wedge(Open\,\mathcal{W}\,\neg\boldsymbol{B_{open}})))\big)$
   (if $Move$ is not true, open the door while $\boldsymbol{B_{open}}$ holds)
4. $\Box(\boldsymbol{B_i}\rightarrow \Diamond Floor_i)$
   (if asked to come or go to the i-th floor, eventually arrive at the floor)

$\boldsymbol{B_{open}}$ and $\boldsymbol{B_i}$ are input propositions and the other propositions are output propositions. If $\boldsymbol{B_{open}}$will be true forever after some state where both $\neg Move$ and $Floor_i$ hold, and $\boldsymbol{B_{j(\neq i)}}$will be true after this state, $\Diamond Floor_j$ could never

be satisfied. This example shows that a specification is not realizable if for some infinite input sequence a $\Diamond$-formula has no opportunity to hold. This example shows that a specification is not stepwisely satisfiable if it could require a response depending on the future sequences of requests.

4 Decision Procedure

This section gives the decision procedure of stepwise satisfiability, which was given in [12]. This procedure is sound and complete. This procedure is based on the tableau method for PLTL [9].

4.1 Tableau Method

A tableau is a directed graph $T = \langle N, E\rangle$ constructed from a given specification. N is a finite set of nodes. Each node is a set of formulas. E is a finite set of edges. Each edge is a pair of nodes. n_2 is reachable from n_1 in a tableau $\langle N, E\rangle$ if and only if $\langle n_1, a_1\rangle, \langle a_1, a_2\rangle, \cdots \langle a_k, n_2\rangle \in E$.

Definition 5 (Decomposition Procedure). *A decomposition procedure takes a set S of formulas as input and produces a set Σ of sets of formulas.*

1. *Put $\Sigma = \{S\}$.*
2. *Repeatedly apply one of steps (a) – (e) to all the formulas f_{ij} in all the sets $S_i \in \Sigma$ according to the type of the formulas until no step will change Σ. In the following, $f_1 \mathcal{W}^* f_2$ and $\neg(f_1 \mathcal{W}^* f_2)$ are called marked formula. The marks represent that the marked formulae have been applied by the decomposition produce.*
   (a) If f_{ij} is $\neg\neg f$, replace S_i with the following set: $(S_i - \{f_{ij}\}) \cup \{f\}$.
   (b) If f_{ij} is $f_1 \wedge f_2$, replace S_i with the following set: $(S_i - \{f_{ij}\}) \cup \{f_1, f_2\}$.
   (c) If f_{ij} is $\neg(f_1 \wedge f_2)$, replace S_i with the following two sets:
   $(S_i - \{f_{ij}\}) \cup \{\neg f_1\}$, $(S_i - \{f_{ij}\}) \cup \{\neg f_2\}$.
   (d) If f_{ij} is $f_1 \mathcal{W} f_2$, replace S_i with the following two sets:
   $(S_i - \{f_{ij}\}) \cup \{f_2\}$, $(S_i - \{f_{ij}\}) \cup \{f_1, \neg f_2, f_1 \mathcal{W}^* f_2\}$.
   (e) If f_{ij} is $\neg(f_1 \mathcal{W} f_2)$ replace S_i with the following two sets:
   $(S_i - \{f_{ij}\}) \cup \{\neg f_1, \neg f_2\}$, $(S_i - \{f_{ij}\}) \cup \{f_1, \neg f_2, \neg(f_1 \mathcal{W}^* f_2)\}$.

Definition 6. *A node n of a tableau $\langle N, E\rangle$ is closed if and only if one of the following conditions is satisfied:*

- *n contains both an atomic proposition and its negation.*
- *n contains an eventuality formula*[1] *and all reachable unclosed nodes from n contain the same eventuality formula.*
- *n cannot reach any unclosed node.*

[1] An eventuality formula is a formula of the form $\neg(f_1 \mathcal{W} f_2)$.

The following describes the tableau construction procedure that repeatedly uses the decomposition procedure. The tableau construction procedure takes a PLTL formula φ as input and produces a tableau $T = \langle N, E\rangle$. In the procedure, a function $temporal(n)$ is used and defined as follows.

$temporal(n) = \{f_1 \mathcal{W} f_2 \mid f_1 \mathcal{W}^* f_2 \in n\} \cup \{\neg(f_1 \mathcal{W} f_2) \mid \neg(f_1 \mathcal{W}^* f_2) \in n\}$

Definition 7 (Tableau Construction Procedure). *The tableau construction procedure takes a formula φ as input and produces a tableau of φ.*

1. *Put $N = \{START, \{\varphi\}\}$ and $E = \{\langle START, \{\varphi\}\rangle\}$ ($START$ is the initial node).*
2. *Repeatedly apply steps (a) and (b) to $T = \langle N, E\rangle$ until T no longer changes.*
   (a) *(decomposition of states) Apply the following three steps to all the nodes $n_i \in N$ to which these steps have not been applied yet.*
      i. *Apply the decomposition procedure to n_i (Σ_{n_i} is defined to be the output of the decomposition procedure)*
      ii. *Replace E with $(E \cup \{\langle m, m'\rangle | \langle m, n_i\rangle \in E$ and $m' \in \Sigma_{n_i}\})$ $-\{\langle m, n_i\rangle | m \in N\}$,*
      iii. *Replace N with $(N - n_i) \cup \Sigma_{n_i}$.*
   (b) *(transition of states) Apply the following two steps to all the nodes $n_i \in N$ to which these steps have not been applied yet.*
      i. *Replace E with $E \cup \{\langle n_i, temporal(n_i)\rangle\}$.*
      ii. *Replace N with $N \cup \{temporal(n_i)\}$.*

In [10,11], it is proved that a formula is satisfiable if and only if the initial node $START$ of the tableau of the formula is unclosed. Thus, the following procedure decides the satisfiability of the formula φ.

Definition 8 (Tableau Method). *The following procedure decides whether a formula φ is satisfiable.*

1. *By Tableau Construction Procedure, construct a tableau of a formula φ*
2. *If the tableau of the formula φ is unclosed, it is concluded that the formula φ is satisfiable. Otherwise, it is concluded that the formula φ is unsatisfiable.*

4.2 Decision Procedure for Stepwise Satisfiability

This subsection describes the decision procedure of stepwise satisfiability. In the decision procedure it is important to to make a tableau deterministic by a set of input and output propositions. At the beginning, some functions are defined for a deterministic tableau.

Definition 9. *Let $T = \langle N, E\rangle$ be a tableau. The function $next(\mathbf{n})$ maps a subset of N to a subset of N, where $\mathbf{n}$ is a subset of N.*

The function $next(\mathbf{n})$ is defined as follows:

$next(\mathbf{n}) \equiv \bigcup_{n \in \mathbf{n}} \{n' \mid \langle n, n'\rangle \in E\}$

The function atm and $\overline{atm}$ map an element of N to a set of atomic propositions. The function atm and $\overline{atm}$ are defined as follows:

$atm(n) \equiv \{f \mid f \in n \text{ and } f \text{ is atomic formula.}\}$
$\overline{atm}(n) \equiv \{f \mid \neg f \in n \text{ and } f \text{ is atomic formula.}\}$
For an element of $\mathcal{N}$*, the function atm and* $\overline{atm}$ *are defined as follows:*
$atm(\mathbf{n}) \equiv \bigcup_{n \in \mathbf{n}} \{f \mid f \in n \text{ and } f \text{ is atomic formula.}\}$
$\overline{atm}(\mathbf{n}) \equiv \bigcup_{n \in \mathbf{n}} \{f \mid \neg f \in n \text{ and } f \text{ is atomic formula.}\}$
Suppose that $a \subseteq P$ *and* $next(\mathbf{n})/a$ *is defined as follows:*
$next(\mathbf{n})/a \equiv \{n | n \in next(\mathbf{n}),\ \overline{atm}(n) \cap a = \emptyset \text{ and } atm(n) \cap (P - a) = \emptyset\}$

Next, a deterministic tableau is defined.

Definition 10 (Deterministic tableau). *Let* $\langle N, E \rangle$ *be a tableau of specification* φ*.* $\langle \mathcal{N}, \mathcal{E} \rangle$ *is a deterministic tableau of* φ *and a set* P *of atomic propositions if and only if the following conditions are satisfied.*

- $\mathcal{N}$ *is a set of tableau node sets, that is* $\mathcal{N} \subseteq 2^N$*.*
- *Every element of* $\mathcal{N}$ *is not an empty set.*
- $\mathcal{E}$ *is a set of* $\langle \mathbf{n_1}, a, \mathbf{n_2} \rangle$ *such that* $\mathbf{n_1}, \mathbf{n_2} \in \mathcal{N}$ *and* $a \subseteq P$*.*
- *If* $\langle \mathbf{n}', a, \mathbf{n} \rangle \in \mathcal{E}$*,* $a \cap \overline{atm}(\mathbf{n}) = \emptyset$ *and* $(P - a) \cap atm(\mathbf{n}) = \emptyset$
- *If* $\langle \mathbf{n}_1, a, \mathbf{n}_2 \rangle \in \mathcal{E}$*, for* $n \in \mathbf{n}_2$*, there is* $n' \in \mathbf{n}_1$ *such that* $\langle n', n \rangle \in E$*.*

The following defines a procedure for constructing a deterministic tableau that is used in the decision procedure of stepwise satisfiability.

Definition 11 (Tableau Deterministic Procedure). *The following procedure constructs a deterministic tableau* $\mathcal{T} = \langle \mathcal{N}, \mathcal{E} \rangle$ *of specification of* φ *and a set* P *of atomic propositions.*

1. *Construct tableau* $\langle N, E \rangle$ *of* φ *by the tableau construction procedure.*
2. *Set* $\mathcal{N} = \{\{START\}\}$ *and* $\mathcal{E} = \emptyset$*.*
3. *Repeat the following step until* $\mathcal{T}$ *no longer changes.*
   For $\mathbf{n} \in \mathcal{N}$ *and* $a \subseteq P$*, if* $next(\mathbf{n})/a \neq \emptyset$*, add* $next(\mathbf{n})/a$ *into* $\mathcal{N}$ *and* $\langle \mathbf{n}, a, next(\mathbf{n})/a \rangle$ *into* $\mathcal{E}$*, where* $next(\mathbf{n})/a \equiv \{n \in next(\mathbf{n}) \mid a \cap \overline{atm}(n) = \emptyset, (P - a) \cap atm(n) = \emptyset\}$*.*

By using a deterministic tableau, it is possible to decide whether there is a reactive system RS of φ such that RS behaves for an input event at any time, or whether there is an infinite output event sequence for any infinite input event sequence. However, it is impossible to decide satisfiability of φ using a deterministic tableau and to check the satisfiability of φ requires to examine each part of a tableau included in the deterministic tableau.

Definition 12 (Decision Procedure of Stepwise Satisfiability).

1. *By the tableau deterministic procedure, construct a deterministic tableau* $\langle \mathcal{N}, \mathcal{E} \rangle$ *of specification* φ *and a set* $\mathcal{X} \cup \mathcal{Y}$ *of input and output propositions.*
2. *Repeat the following operation until* $\langle \mathcal{N}, \mathcal{E} \rangle$ *no longer changes.*
   For $\mathbf{n} \in \mathcal{N}$ *and* $a \subseteq \mathcal{X}$*, if there are no* $\mathbf{n}' \in \mathcal{N}$ *and* $b \subseteq \mathcal{Y}$ *such that* $\langle \mathbf{n}, a \cup b, \mathbf{n}' \rangle \in \mathcal{E}$*, delete* $\mathbf{n}$ *from* $\mathcal{N}$*. Delete elements such as* $\langle \mathbf{n}, c, \mathbf{n}' \rangle$ *or* $\langle \mathbf{n}', c, \mathbf{n} \rangle$ *from* $\mathcal{E}$*.*
3. *If* $\mathcal{N}$ *is not an empty set, this procedure determines that* φ *is stepwisely satisfiable. Otherwise, this procedure determines that* φ *is not stepwisely satisfiable.*

5 Proof System

This section gives a proof system for unrealizability of open reactive system specification. This proof system is a sequent-style natural deduction. Before giving several formal definitions, this section informally explains the meanings of sequent and several symbols. This proof system uses sequent $\Gamma, \Delta^\dagger \vdash W$ where Γ is a set of input proposition formula $\Delta^\dagger$ is a set of mark formulas such $A^\dagger$ and W is a formula. If $\Delta^\dagger$ is an empty set, this sequent denotes that there exists an open reactive system RS such that for each sequence $\tilde{a}$ of output event sets, if $\tilde{a} \models \bigwedge \Gamma$ ($\bigwedge \Gamma$ is a conjunction of all elements of Γ), $behave_{RS}(\tilde{a}) \models W$; if $\Delta^\dagger$ is not an empty set, this sequent denotes that there exists an open reactive system RS such that for each sequence $\tilde{a}$ of output event sets, if $\tilde{a} \models \bigwedge \Gamma$ and if there exists V such that $\tilde{a} \models V$ and $V^\dagger \in \Delta^\dagger$, $behave_{RS}(\tilde{a}) \models W$.

To check a specification φ requires to make a proof that begins with $\vdash \varphi$ and whose conclusion is contradiction. While constructing a proof, several input formulas (a formula consisting of input propositions) move from the right side to left side of sequent. If the formulas on the right side of a sequent is contradiction and the formulas on the left side are satisfiable, it is concluded that φ is unrealizable. If there are several marked formulas on the left side, it is necessary only to check whether all unmarked formulas and one of marked formulas are satisfiable, even if a set of marked formulas are inconsistent. The marked formulas represent the input formulas which can hold in one of possible future states, therefore, a set of the marked formulas can be inconsistent. On the other hand, unmarked formulas must be satisfiable in the same state such as current state or future state. This is a reason why the proof system uses mark. The example proof in section 5.2 helps to understand this reason. The following gives formal definitions.

Definition 13. *$A^\dagger$ is defined to be mark formula where A is a formula. $\Delta^\dagger$ is defined to a set of marked input formulas where $\Delta^\dagger = \{A^\dagger \mid A \in \Delta\}$.*

Definition 14. *$\Gamma, \Delta^\dagger \vdash W$ is defined to be sequent where Γ is a set of input formulas, $\Delta^\dagger$ is a set of marked input formulas and W is a formula.*

A sequent $\Gamma, \Delta^\dagger \vdash W$ holds with respect to an open reactive system RS if and only if the following condition holds.

- *In the case that $\Delta^\dagger$ is an empty set, for each sequence $\tilde{a}$ of output event sets, if $\tilde{a} \models \Gamma$, $behave_{RS}(\tilde{a}) \models W$.*
- *In the case that $\Delta^\dagger$ is not an empty set, for each sequence $\tilde{a}$ of output event sets, if there is $V \in \Delta$ such that $\tilde{a} \models \Gamma \cup \{V\}$, $behave_{RS}(\tilde{a}) \models W$.*

Definition 15. *The inference rules are defined as follows. In the inference rules, $X, X_1, X_2, \ldots$ are formulas consisting of only input propositions and A, B, C are arbitrary formulas; a temporal formula is defined to be a formula including temporal operator such as $\Box$, $\Diamond$, $\mathcal{W}$ and $\bigcirc$. The proof system in this paper uses*

"Next operator", but it is not defined in the syntax of PLTL. This paper defines next operator ($\bigcirc$) as usual.

$$\frac{\Gamma, \Delta^\dagger \vdash A \wedge B}{\Gamma, \Delta^\dagger \vdash A}\ \wedge E1 \qquad \frac{\Gamma, \Delta^\dagger \vdash A \wedge B}{\Gamma, \Delta^\dagger \vdash B}\ \wedge E2 \qquad \frac{\Gamma_1, \Delta_1^\dagger \vdash A \quad \Gamma_2, \Delta_2^\dagger \vdash B}{\Gamma_1, \Gamma_2, \Delta_1^\dagger, \Delta_2^\dagger \vdash A \wedge B}\ \wedge I$$

$$\frac{\Gamma, \Delta^\dagger \vdash A \vee B \quad \begin{array}{c}[\Gamma, \Delta^\dagger \vdash A] \\ \vdots \\ \Gamma_1, \Delta_1^\dagger \vdash C\end{array} \quad \begin{array}{c}[\Gamma, \Delta^\dagger \vdash B] \\ \vdots \\ \Gamma_2, \Delta_2^\dagger \vdash C\end{array}}{\Gamma_1, \Gamma_2, \Delta_1^\dagger, \Delta_2^\dagger \vdash C}\ \vee E^\star \qquad \frac{\Gamma, \Delta^\dagger \vdash A}{\Gamma, \Delta^\dagger \vdash A \vee B}\ \vee I1 \qquad \frac{\Gamma, \Delta^\dagger \vdash B}{\Gamma, \Delta^\dagger \vdash A \vee B}\ \vee I2$$

$$\frac{\Gamma_1, \Delta_1^\dagger \vdash A \quad \Gamma_2, \Delta_2^\dagger \vdash \neg A}{\Gamma_1, \Gamma_2, \Delta_1^\dagger, \Delta_2^\dagger \vdash \bot}\ \bot I \qquad \frac{\Gamma, \Delta^\dagger \vdash \bot}{\Gamma, \Delta^\dagger \vdash A}\ \bot E$$

$$\frac{\Gamma, \Delta^\dagger \vdash A \mathcal{W} B}{\Gamma, \Delta^\dagger \vdash B \vee A \wedge \neg B \wedge \bigcirc(A \mathcal{W} B)}\ \mathcal{W}1 \qquad \frac{\Gamma, \Delta^\dagger \vdash \neg(A \mathcal{W} B)}{\Gamma, \Delta^\dagger \vdash (\neg A \wedge \neg B) \vee (A \wedge \neg B \wedge \bigcirc \neg(A \mathcal{W} B))}\ \mathcal{W}2$$

$$\frac{\Gamma, \Delta^\dagger \vdash A}{\Gamma, \Delta^\dagger \vdash \Diamond A}\ \Diamond I \qquad \frac{\Gamma, \Delta^\dagger \vdash A}{\Box\Gamma, \Delta^\dagger \vdash \Box A}\ \Box I^\star \qquad \frac{\Gamma, \Delta^\dagger \vdash X}{\Gamma, \Delta^\dagger, \neg X \vdash \bot}\ LM1 \qquad \frac{\Gamma, \Delta^\dagger, \vdash \bigcirc X}{\Gamma, \Delta^\dagger, \bigcirc \neg X^\dagger \vdash \bot}\ LM2^{\star\star}$$

$$\frac{\Gamma, \Delta^\dagger, X_1 \vee X_2 \vdash A}{\Gamma, \Delta^\dagger, X_1 \vdash A}\ \vee LE1 \qquad \frac{\Gamma, \Delta^\dagger, X_1 \vee X_2 \vdash A}{\Gamma, \Delta^\dagger, X_2 \vdash A}\ \vee LE2 \qquad \frac{\Gamma, \Delta^\dagger, \neg(X_1 \vee X_2) \vdash A}{\Gamma, \Delta^\dagger, \neg X_1, \neg X_2 \vdash A}\ \vee LE3$$

$$\frac{\Gamma, \Delta^\dagger, X_1 \wedge X_2 \vdash A}{\Gamma, \Delta^\dagger, X_1, X_2 \vdash A}\ \wedge LE1 \qquad \frac{\Gamma, \Delta^\dagger, \neg(X_1 \wedge X_2) \vdash A}{\Gamma, \Delta^\dagger, \neg X_1 \vee \neg X_2 \vdash A}\ \wedge LE2$$

$$\frac{\Gamma, \Delta^\dagger, X_1 \mathcal{W} X_2 \vdash A}{\Gamma, \Delta^\dagger, X_2 \vee X_1 \wedge \neg X_2 \wedge \bigcirc(X_1 \mathcal{W} X_2) \vdash A}\ \mathcal{W}L1$$

$$\frac{\Gamma, \Delta^\dagger, \neg(X_1 \mathcal{W} X_2) \vdash A}{\Gamma, \Delta^\dagger, (\neg X_1 \wedge \neg X_2) \vee (X_1 \wedge \neg X_2 \wedge \bigcirc \neg(X_1 \mathcal{W} X_2)) \vdash A}\ \mathcal{W}L2$$

$$\frac{\Gamma, \Delta^\dagger, \neg\neg X \vdash A}{\Gamma, \Delta^\dagger, X \vdash A}\ DNLE \qquad \frac{\Gamma, \Delta^\dagger, X \vdash A}{\Gamma, \Delta^\dagger, \Box X \vdash A}\ \Box LI$$

$$\frac{\Gamma, \Delta^\dagger, X, X \vdash A}{\Gamma, \Delta^\dagger, X \vdash A}\ CL \qquad \frac{\Gamma, \Delta^\dagger, \neg \bigcirc X \vdash A}{\Gamma, \Delta^\dagger, \bigcirc \neg X \vdash A}\ \neg LN$$

$\star$ *Hypotheses $\Gamma, \Delta^\dagger \vdash A$ and $\Gamma, \Delta^\dagger \vdash B$ are discarded by using $\vee E$.*
$\star\star$ *There is no hypothesis or all hypotheses are of the form $\Box A$.*

Definition 16. *A proof beginning with formula φ is defined to be a sequence $S_1, S_2 \ldots S_n$ of sequent satisfying the following conditions.*

1. *"Implication" is expressed by "not" and "disjunction", that is, $A \to B$ have to be converted into $\neg A \vee B$.*
2. *Every S_k $(1 \leq k \leq n)$ is one of the following*
   - $\vdash \varphi$
   - *an assumption, which should have been discarded by inference rule $\vee E$ in Definition 15.*

- *a conclusion of an inference rule in Definition 15 where the premises of the inference rule must be in* $S_l (1 \leq l \leq k)$.

3. S_k $(1 \leq k \leq n)$ *is a sequent* $\Gamma, \Delta^\dagger \vdash A$ *satisfying following.*
   - *If* Δ *is empty,* Γ *is consistent.*
   - *If* Δ *is not empty, there exists a formula* V *such that* $V \in \Delta$ *and* $\Gamma \cup \{V\}$ *is consistent.*

S_n *is defined to be a conclusion of a proof* $S_1, S_2 \ldots S_n$.

The definition of proof uses concept of consistency of a set of formulas; it is undesirable in the definition of proof system. However, this proof system decides realizability but not satisfiability and realizability is considered to be meta concept rather than satisfiability. It follows that it is allowable to use consistency check in the proof system. Implementation of this proof system does not always have to check consistency.

Let us explain the outline of how to prove unrealizability of a formula φ; we try to make a proof beginning with formula φ. If a sequent $\Gamma, \Delta^\dagger \vdash \bot$ is a conclusion of a proof and $\Gamma \cup \{V\}$ is satisfiable for some element V of Δ, it is concluded that φ is unrealizable; in the case that Δ is an empty set, if Γ is satisfiable, it is concluded that φ is unrealizable. The following gives this definition.

Definition 17 (Unrealizability Decision). *Suppose that a sequent* $\Gamma, \Delta^\dagger \vdash \bot$ *is a conclusion of a proof beginning with* φ. *In the case that* $\Delta^\dagger$ *is not an empty set, if* $\Gamma \cup \{V\}$ *is satisfiable for some element* V *of* Δ, *it is concluded that* φ *is unrealizable; in the case that* Δ *is an empty set, if* Γ *is satisfiable, it is concluded that* φ *is unrealizable.*

5.1 Soundness

This subsection proves soundness of the realizability proof system

Theorem 1. *If the proof system decides that* φ *is unrealizable,* φ *is unrealizable.*

Theorem 1 is proved by induction on the structure of proof, in which the following lemma is used.

Lemma 1. *Suppose that* $S_1, S_2 \ldots S_n$ *is a proof beginning with* φ. *If there exists an open reactive system RS satisfying the following conditions, the sequent* S_n *holds with respect to RS.*

- $behave_{RS}(\tilde{a}) \models \varphi$ *for each sequence* $\tilde{a}$ *of output event sets.*
- *If* S_k *is a hypothesis that is not discarded,* S_k *holds with respect to RS.*

Proof of Lemma 1: This lemma is proved by induction on the structure of proof. This paper shows induction step of inference rule LM because of space limitation. All induction step proofs are shown by the same way.

– $LM1$
By induction hypothesis, $\Gamma, \Delta^\dagger \vdash X$ holds with respect to RS satisfying the conditions of Lemma 1. Suppose that Δ is not empty. By the definition, for each sequence $\tilde{a}$ of output event sets, if there exists $V \in \Delta$ such that $\tilde{a} \models \Gamma \cup \{V\}$, $behave_{RS}(\tilde{a}) \models X$. For each sequence $\tilde{a}$ of output event sets, if there exists $V \in \Delta$ such that $\tilde{a} \models \Gamma \cup \{\neg X, V\}$, $behave_{RS}(\tilde{a}) \models \neg X$ because X consists of input propositions; it follows that $behave_{RS}(\tilde{a}) \models \bot$. This lemma can be also proved in the case that Δ is empty.

This lemma can be proved in the case of the other inference rules similarly. ■

Now, Lemma 1 proves Theorem 1. Suppose that unrealizability decision decides that a formula φ is unrealizable. The definition of unrealizability judgment obtains a conclusion $\Gamma, \Delta^\dagger \vdash \bot$ in a proof beginning with φ, and there is $V \in \Delta$ such that $\Gamma \cup \{V\}$ is satisfiable in the case that $\Delta^\dagger$ is not empty, and Γ is satisfiable in the case that $\Delta^\dagger$ is empty. By Lemma 1, if there is an open reactive system such that $\vdash \varphi$ holds with respect to RS, $\Gamma, \Delta^\dagger \vdash \bot$ holds with respect to RS. However, there exists a sequence $\tilde{a}$ of input event sets such that $\tilde{a} \models \Gamma \cup \{V\}$ (Γ if $\Delta^\dagger$ is empty) for some element V of Δ. By the definition of sequent, there should be an open reactive system RS such that $behave_{RS}(\tilde{a}) \models \bot$, however, there is not such an open reactive system RS; it is inconsistent. By reductio ad absurdum, there is no an open reactive system RS which $\vdash \varphi$ does not hold with respect to, and φ can prove to be unrealizable.

5.2 Proof Examples

This subsection shows several proof examples of realizability proof system described in the previous section. In the following, $x, x_1, x_2, \cdots$ represents input propositions and $y, y_1, y_2, \cdots$ represents output propositions. Figure 1 gives several inference rules of in order to show examples easily. These inference rules can be derived from those of Definition 15.

$$\frac{\Gamma, \Delta^\dagger \vdash \Box A}{\Gamma, \Delta^\dagger \vdash A}\ \Box E \qquad \frac{\Gamma, \Delta^\dagger \vdash X \vee A}{\Gamma, \Delta^\dagger, \neg X \vdash A}\ LM3$$

Fig. 1. Additional inference rules

1. $\Box(x_1 \to y) \wedge \Box(x_2 \to \Diamond\neg y)$
This formula is not realizable because if x_2 are true at some point and x_1 is always false, y is always true while $\neg y$ have to be true at some future point. We start the following proof by $\Box(\neg x_1 \vee y)$ and $\Box(\neg x_2 \vee \Diamond\neg y)$ which are equal to $\Box(x_1 \to y) \wedge \Box(x_2 \to \Diamond\neg y)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \Box(\neg x_1 \vee y)}{\vdash \neg x_1 \vee y}\ \Box E}{\neg\neg x_1 \vdash y}\ LM3}{x_1 \vdash y}\ DNLE}{\Box x_1 \vdash \Box y}\ \Box I \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \Box(\neg x_2 \vee \Diamond\neg y)}{\vdash \neg x_2 \vee \Diamond\neg y}\ \Box E}{\neg\neg x_2 \vdash \Diamond\neg y}\ LM3}{x_2 \vdash \Diamond\neg y}\ DNLE}{x_2 \vdash \neg\Box y}\ (Abbreviation)}{\Box x_1, x_2 \vdash \bot}\ \bot I$$

2. $x_1 \mathcal{W} x_2 \leftrightarrow y$
   This formula is not realizable; when x_1 and $\neg x_2$ are true, the truth value of y depends on the future truth values of x_1 and x_2. Suppose that an open reactive system makes y true at first time. If $\neg x_1$ and $\neg x_2$ are true at next time, this formula does not holds at first time because $x_1 \mathcal{W} x_2$ is false and y is true. On the other hand, suppose that an open reactive system makes y false at first time. If x_2 are true at next time, this formula does not holds at first time because $x_1 \mathcal{W} x_2$ is true and y is false.

   After an open reactive system decides a truth value of y, the environment of the open reactive system can behave in order to this formula become false. To prove unrealizability of such formulas uses the inference rule $LM2$. The proof in Figure 2 begins with $x_1 \mathcal{W} x_2 \vee \neg y$ and $\neg(x_1 \mathcal{W} x_2) \vee y$, which are equal to $x_1 \mathcal{W} x_2 \leftrightarrow y$. All formulas in the left side of the conclusion sequent are inconsistent because $\neg\bigcirc x_1 \mathcal{W} x_2^{\dagger}$ and $\neg\bigcirc\neg(x_1 \mathcal{W} x_2)^{\dagger}$ are included. However, this proof system deals with marked formulas separately and checks satisfiability of only following two sets of formulas:

$$\{\neg x_2, \neg(\neg x_2 \wedge \neg x_1), \neg\bigcirc x_1 \mathcal{W} x_2\} \text{ and } \{\neg x_2, \neg(\neg x_2 \wedge \neg x_1), \neg\bigcirc\neg(x_1 \mathcal{W} x_2)\}$$

   Each of these two sets is consistent, but these two sets suggest that in some future any system behaviors will be inconsistent to satisfy $x_1 \mathcal{W} x_2 \leftrightarrow y$. This proof has several inference rules that can deduce unrealizability of a specification based on this inconsistency.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\vdash x_1 \mathcal{W} x_2]}{\vdash x_2 \vee x_1 \wedge \neg x_2 \wedge \bigcirc x_1 \mathcal{W} x_2}\,\mathcal{W}1}{\neg x_2 \vdash x_1 \wedge \neg x_2 \wedge \bigcirc x_1 \mathcal{W} x_2}\,LM3}{\neg x_2 \vdash \bigcirc x_1 \mathcal{W} x_2}\,\wedge E2}{\neg x_2, \neg\bigcirc x_1 \mathcal{W} x_2^{\dagger} \vdash \bot}\,LM2}{\vdash x_1 \mathcal{W} x_2 \vee \neg y \;\; \neg x_2, \neg\bigcirc x_1 \mathcal{W} x_2^{\dagger} \vdash \neg y \;[\vdash \neg y]}\,\bot E \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[\vdash \neg(x_1 \mathcal{W} x_2)]}{\vdash \neg x_2 \wedge \neg x_1 \vee \neg x_2 \wedge x_1 \wedge \bigcirc\neg(x_1 \mathcal{W} x_2)}\,\mathcal{W}2}{\neg(\neg x_2 \wedge \neg x_1) \vdash \neg x_2 \wedge x_1 \wedge \bigcirc\neg(x_1 \mathcal{W} x_2)}\,LM3}{\neg(\neg x_2 \wedge \neg x_1) \vdash \bigcirc\neg(x_1 \mathcal{W} x_2)}\,\wedge E2}{\neg(\neg x_2 \wedge \neg x_1), \neg\bigcirc\neg(x_1 \mathcal{W} x_2)^{\dagger} \vdash \bot}\,LM2}{\vdash \neg(x_1 \mathcal{W} x_2) \vee y \;\; \neg(\neg x_2 \wedge \neg x_1), \neg\bigcirc\neg(x_1 \mathcal{W} x_2)^{\dagger} \vdash y \;[\vdash y]}\,\bot E$$

$$\dfrac{\neg x_2, \neg\bigcirc x_1 \mathcal{W} x_2^{\dagger} \vdash \neg y \qquad \neg(\neg x_2 \wedge \neg x_1), \neg\bigcirc\neg(x_1 \mathcal{W} x_2)^{\dagger} \vdash y}{\neg x_2, \neg(\neg x_2 \wedge \neg x_1), \neg\bigcirc x_1 \mathcal{W} x_2^{\dagger}, \neg\bigcirc\neg(x_1 \mathcal{W} x_2)^{\dagger} \vdash \bot}\,\vee E\;\bot I$$

Fig. 2. The proof of $x_1 \mathcal{W} x_2 \leftrightarrow y$

6 Implementation

This section gives the implementation of stepwise satisfiability checker based on the tableau method and the proof system described in the previous section. The tableau based decision procedure of stepwise satisfiability takes too much time. Thus, the following stepwise satisfiability checking procedure checks realizability of a specification in creating nodes of tableau; proof system checking of unrealizability and creating tableau are performed simultaneously. If the proof system proves unrealizability of a specification, the stepwise satisfiability of checking procedure can decide unrealizability of the specification fast without creating

tableau. In the following, an element $\mathbf{n}$ of $\mathcal{N}$ of deterministic tableau $\langle \mathcal{N}, \mathcal{E} \rangle$ is interpreted as formula $\bigvee_{n \in \mathbf{n}} \bigwedge_{f \in n} f$.

Definition 18 (Stepwise Satisfiability Checking Procedure). *This procedure checks stepwise satisfiability of φ.*

1. *Check unrealizability of φ by the proof system within a predefined time or memory.*
2. *If the proof system decides unrealizability of φ, this procedure ends and decides that φ is unrealizable.*
3. *Create tableau $T = \langle N, E \rangle$ of φ.*
4. *Let $\mathcal{T} = \langle \mathcal{N}, \mathcal{E} \rangle$ be a tableau where $\mathcal{N} = \{\{START\}\}$ and $\mathcal{E} = \emptyset$*
5. *Repeat the following step if $\mathcal{N}$ has the an element $\mathbf{n}$ such that $\langle \mathbf{n}, a, \mathbf{n}' \rangle \notin \mathcal{E}$. Otherwise, this procedure decides that φ is stepwisely satisfiable.*
   (a) *For each $a \subseteq P$, if $next(\mathbf{n})/a \neq \emptyset$, check unrealizability of a formula of $next(\mathbf{n})/a$ by the proof system. If the proof system does not decide that a formula of $next(\mathbf{n})/a$ is unrealizable, add $next(\mathbf{n})/a$ into $\mathcal{N}$ and $\langle \mathbf{n}, a, next(\mathbf{n})/a \rangle$ into $\mathcal{E}$, where $next(\mathbf{n})/a \equiv \{n \in next(\mathbf{n}) \mid a \cap \overline{atm}(n) = \emptyset, (P - a) \cap atm(n) = \emptyset\}$.*
   (b) *For $\mathbf{n} \in \mathcal{N}$ and $a \subseteq \mathcal{X}$, if there are no $\mathbf{n}' \in \mathcal{N}$ and $b \subseteq \mathcal{Y}$ such that $\langle \mathbf{n}, a \cup b, \mathbf{n}' \rangle \in \mathcal{E}$, delete $\mathbf{n}$ from $\mathcal{N}$. Delete elements such as $\langle \mathbf{n}, c, \mathbf{n}' \rangle$ or $\langle \mathbf{n}', c, \mathbf{n} \rangle$ from $\mathcal{E}$.*
   (c) *If $\{START\} \notin \mathcal{N}$, this procedure decides that φ is unrealizable.*

This procedure is different from the tableau based stepwise satisfiability decision procedure. Before adding a new node of deterministic tableau This procedure checks unrealizability of it by proof system. The usage of proof system can omit cost of deterministic tableau and reduce the size of deterministic tableau. This implementation uses MiniSat for satisfiability checking [3,13].

7 Experiment

This section evaluates the implemented stepwise satisfiability checker by experiments. This experiments uses the following PC in which OS is Vine Linux 4.1, CPU is Intel(R) Pentium(R) Dual-Core CPU E5200 2.50GHz and memory size is 4GB. In the following, x, x_1, x_2, ... are input propositions and y,y_1, y_2, ... are output propositions.

(1) The following formulas appear in Section 5.2 They are unrealizable [12].
   1. $\Box(x_1 \rightarrow y) \wedge \Box(x_2 \rightarrow \neg y)$
   2. $\Box(x_1 \rightarrow y) \wedge \Box(x_2 \rightarrow \neg y)$
   3. $(x_1 \mathcal{W} x_2) \rightarrow y) \wedge (y \rightarrow (x_1 \mathcal{W} x_2)$
   4. $(\Diamond x \rightarrow y) \wedge (y \rightarrow \Diamond x)$
   5. $(\Box x \rightarrow \Diamond y) \wedge (\Diamond y \rightarrow \Box x)$

   The result of the stepwise satisfiability checker is that all formulas are unrealizable. It takes less than one second and the proof system decides unrealizability of these formulas in the procedure.

(2) The following temporal formula is a specification of a three floor elevator control system. The proposed procedure cannot decide that specification is stepwisely satisfiable or unrealizable because checking this specification requires a large size of deterministic tableau.

$\Box((y_1 \wedge \neg y_2 \wedge \neg y_3) \vee (y_2 \wedge \neg y_1 \wedge \neg y_3) \vee (y_3 \wedge \neg y_1 \wedge \neg y_2))$
$\Box(x_1 \rightarrow (\Diamond y_1 \wedge y_7 \mathcal{W}(y_1 \wedge y_7))) \wedge \Box((y_1 \wedge y_7) \rightarrow (y_5 \wedge y_1 \mathcal{W} y_4))$
$\Box((y_1 \wedge y_4) \rightarrow (\neg y_7 \mathcal{W} x_1)) \wedge \Box((y_1 \wedge \neg y_7) \rightarrow \neg y_5))$
$\Box((y_1 \wedge y_9) \rightarrow \neg(\neg y_2 \mathcal{W} y_3)) \wedge \Box((x_2 \rightarrow \Diamond y_2) \wedge \neg y_8 \mathcal{W}(y_2 \wedge y_8))$
$\Box((y_2 \wedge y_8) \rightarrow (y_5 \wedge y_2 \mathcal{W} y_4)) \wedge \Box((y_2 \wedge \neg y_8) \rightarrow \neg y_5)$
$\Box((y_2 \wedge y_4) \rightarrow (\neg y_8 \mathcal{W} x_2)) \wedge \Box(x_3 \rightarrow (\Diamond y_3 \wedge (y_9 \mathcal{W}(y_3 \wedge y_9))))$
$\Box((y_3 \wedge y_9) \rightarrow (y_5 \wedge y_3 \mathcal{W} y_4)) \wedge \Box((y_3 \wedge \neg y_9) \rightarrow \neg y_5)$
$\Box((y_3 \wedge y_4) \rightarrow (\neg y_9 \mathcal{W} x_3)) \wedge \Box((y_3 \wedge y_7) \rightarrow \neg(\neg y_2 \mathcal{W} y_1))$
$\Box(y_5 \rightarrow (\neg y_4 \mathcal{W} \neg y_5)) \wedge \Box(\neg y_5 \rightarrow (y_4 \mathcal{W} y_5))$
$\Box(y_5 \rightarrow \Diamond y_{10}) \wedge \Box((x_4 \wedge \neg y_{10}) \rightarrow y_6)$
$\Box(y_{10} \rightarrow \neg y_5) \wedge \Box((x_5 \wedge \neg y_6) \rightarrow \neg y_5) \wedge \Box((y_6 \wedge \neg y_4) \rightarrow y_5)$

(3) Within 1 second, this procedure can check unrealizability of the specification that is obtained by exchanging input and output propositions in the previous specification. This fast decision depends on the proposed proof system. Although this result does not seem inconsistent with the result of (2), the result of (3) does not directly deduce the property of (2).

(4) The specification of (2) is difficult for unrealizability or stepwise satisfiability check. Some parts of the previous specification are checked in this experiment.

$\Box((y_1 \wedge \neg y_2 \wedge \neg y_3) \vee (y_2 \wedge \neg y_1 \wedge \neg y_3) \vee (y_3 \wedge \neg y_1 \wedge \neg y_2))$
$\Box(x_1 \rightarrow (\Diamond y_1 \wedge y_7 \mathcal{W}(y_1 \wedge y_7))) \wedge \Box((y_1 \wedge y_7) \rightarrow (y_5 \wedge y_1 \mathcal{W} y_4))$
$\Box((y_1 \wedge y_4) \rightarrow (\neg y_7 \mathcal{W} x_1))$

This procedure decides that this formula is stepwisely satisfiable in 2567 seconds.

8 Conclusion

This paper proposed a proof system of unrealizability of open reactive system specification and proved soundness of the proof system. This proof system can decide unrealizability of a specification fast, but it cannot always decide unrealizability. This paper implemented stepwise satisfiability checking procedure, which consists of the tableau method and the proof system. This stepwise satisfiability checking procedure can decide that a specification is unrealizable or that a specification is stepwisely satisfiable. This procedure is incomplete for unrealizability checking or stepwise satisfiability checking. However, this procedure can decide the property of specification realizability. The experiment showed that the procedure can decide very fast that several specifications are unrealizable or stepwisely satisfiable. One of the future works is to improve the decision procedure for unrealizability or stepwise satisfiability.

References

1. Abadi, M., Lamport, L., Wolper, P.: Realizable and Unrealizable Specifications of Reactive Systems. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 1–17. Springer, Heidelberg (1989)
2. Bouyer, P., Bozzelli, L., Chevalier, F.: Controller Synthesis for MTL Specifications. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 450–464. Springer, Heidelberg (2006)
3. Eén, N., Mishchenko, A., Sörensson, N.: Applying Logic Synthesis for Speeding Up SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 272–286. Springer, Heidelberg (2007)
4. Harel, D., Pnueli, A.: On the development of reactive systems. In: Logics and Models of Concurrent Systems. NATO Advanced Summer Institutes, vol. F-13, pp. 477–498 (1985)
5. Kupferman, O., Madhusudan, P., Thiagarajan, P.S., Vardi, M.Y.: Open Systems in Reactive Environments: Control and Synthesis. In: Palamidessi, C. (ed.) CONCUR 2000. LNCS, vol. 1877, pp. 92–107. Springer, Heidelberg (2000)
6. Ouaknine, J., Worrell, J.: On the Decidability of Metric Temporal Logic. In: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science, pp. 188–197 (2005)
7. Pnueli, A., Rosner, R.: On the Synthesis of an Asynchronous Reactive Module. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)
8. Pnueli, A., Rosner, R.: On the Synthesis of a Reactive Module. In: Proc. 16th Ann. ACM Symp. on the Principle of Programming Languages, pp. 179–190 (1989)
9. Mori, R., Yonezaki, Y.: Derivation of the Input Conditional Formula from a Reactive System Specification in Temporal Logic. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 567–582. Springer, Heidelberg (1994)
10. Emerson, E.A.: Temporal and Modal Logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. North-Holland, Amsterdam (1990)
11. Wolper, P.: Temporal Logic can be more expressive. Informaition and Control 56, 72–93 (1983)
12. Yoshiura, N.: Decision Procedures for Several Properties of Reactive System Specifications. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 154–173. Springer, Heidelberg (2004)
13. The MiniSat Page, http://minisat.se/
14. Filiot, E., Jin, N., Raskin, J.-F.: An Antichain Algorithm for LTL Realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
15. Duer-Luts, A.: LTL translation improvements in Spot. In: The 5th International Workshop on Verification and Evaluation of Computer and Communication Systems (2011)

Equational Abstraction Refinement for Certified Tree Regular Model Checking

Yohan Boichut[1], Benoit Boyer[4], Thomas Genet[2], and Axel Legay[3]

[1] LIFO - Université Orléans, France
[2] IRISA - Université Rennes 1, France
[3] INRIA - Rennes, France
[4] VERIMAG - Université Joseph Fourier, France

Abstract. *Tree Regular Model Checking* (TRMC) is the name of a family of techniques for analyzing infinite-state systems in which states are represented by trees and sets of states by tree automata. The central problem is to decide whether a set of bad states belongs to the set of reachable states. An obstacle is that this set is in general neither regular nor computable in finite time.

This paper proposes a new CounterExample Guided Abstraction Refinement (CEGAR) algorithm for TRMC. Our approach relies on a new equational-abstraction based completion algorithm to compute a regular overapproximation of the set of reachable states in finite time. This set is represented by $\mathcal{R}_{/E}$-automata, a new extended tree automaton formalism whose structure can be exploited to detect and remove false positives in an efficient manner. Our approach has been implemented in TimbukCEGAR, a new toolset that is capable of analyzing Java programs by exploiting an elegant translation from the Java byte code to term rewriting systems. Experiments show that TimbukCEGAR outperforms existing CEGAR-based completion algorithms. Contrary to existing TRMC toolsets, the answers provided by TimbukCEGAR are certified by Coq, which means that they are formally proved correct.

1 Introduction

Infinite-state models are often used to avoid potentially artificial assumptions on data structures and architectures, e.g. an artificial bound on the size of a stack or on the value of an integer variable. At the heart of most of the techniques that have been proposed for exploring infinite state spaces, is a symbolic representation that can finitely represent infinite sets of states. In this paper, we rely on Tree Regular Model Checking (TRMC) [19,31], and assume that states of the system are represented by trees and sets of states by tree automata. The transition relation of the system is represented by a set of rewriting rules. Contrary to approaches that are dedicated to specific applications, TRMC is generic and expressive enough to describe a broad class of communication protocols [4], various C programs [16] with complex data structures, multi-threaded programs [34], cryptographic protocols [26,28,5], and Java [13].

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 299–315, 2012.

In TRMC, the central objective is to decide whether a set of states representing some state property belongs to the set of reachable states. An obstacle is that this set is in general neither regular nor computable in a finite time. Most existing solutions rely on computing the transitive closure of the transition relation of the systems through heuristic-based semi-algorithms [31,4], or on the computation of some regular abstraction of the set of reachable states [19,16]. While the first approach is precise, it is acknowledged to be ineffective on complex systems. This paper focuses on the second approach.

The first abstraction-based technique for TRMC, *Abstract Tree Regular Model Checking* (ATRMC), was proposed by Bouajjani et al [17,15,16]. ATRMC computes sequences of automata by successive applications of the rewriting relation to the automaton representing the initial set of states. After each computation step, techniques coming from predicate abstraction are used to over-approximate the set of reachable states. If the property holds on the abstraction, then it also holds on the concrete system. Otherwise, a counter-example is detected and the algorithm has to decide if it is a false positive or not. In case of a spurious counter-example, the algorithm refines the abstraction by backward propagation of the set of rewriting rules. The approach, which may not terminate, proceeds in a CounterExample Guided Abstraction Refinement fashion by successive abstraction/refinement until a decision can be taken. The approach has been implemented in a toolset capable, in part, to analyse C programs.

Independently, Genet et al. [24] proposed *Completion* that is another technique to compute an over-approximation of the set of reachable states. Completion exploits the structure of the term rewriting system to add new transitions in the automaton and obtain a possibly overapproximation of the set of one-step successor states. Completion leads to a direct application of rewriting rules to the automaton, while other approaches rely on possibly heavy applications of sequences of transducers to represent this step. Completion alone may not be sufficient to finitely compute the set of reachable states. A first solution to this problem is to plug one of the abstraction techniques implemented in ATRMC. However, in this paper, we prefer another solution that is to apply equational abstraction [33]. There, the merging of states is induced by a set of equations that largely exploit the structure of the system under verification and its corresponding TRS, hence leading to accurate approximations. We shall see that, initially, such equations can easily be derived from the structure of the system. Later, they are refined automatically with our procedure without manual intervention. Completion with equational abstraction has been applied to very complex case studies such as the verification of (industrial) cryptography protocols [26,28] and Java bytecode applications [13]. CEGAR algorithms based on equational-abstraction completion exist [11,12], but are known to be inefficient.

In this paper, we design the first efficient and certified CEGAR framework for equational-abstraction based completion algorithm. Our approach relies on $\mathcal{R}_{/E}$-automata, that is a new tree automaton formalism for representing sets of reachable states. In $\mathcal{R}_{/E}$-automata, equational abstraction does not merge states, but rather links them with rewriting rules labeled with equations.

Such technique is made easy by exploiting the nature of the completion step. During completion steps, such equations are propagated, and the information can be used to efficiently decide whether a set of terms is reachable from the set of initial states. If the procedure concludes positively, then the term is indeed reachable. Else, one has to refine the $\mathcal{R}_{/E}$-automaton and restart the process again.

Our approach has been implemented in TimbukCEGAR. (T)RMC toolsets result from the combination of several libraries, each of them being implemented with thousands of lines of code. It is thus impossible to manually prove that those tools deliver correctly answers. A particularity of TimbukCEGAR is that it is certified. In order to ensure that the whole set of reachable states has been explored, any TRMC technique needs to check whether a candidate overapproximation B is indeed a fixed point, that is if $\mathcal{L}(B) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. Such check has been implemented in various TRMC toolsets, but there is no guarantee that it behaves correct. In [20], a checker for tree automata completion was designed and proved correct using the Coq [9] proof assistant. Any automaton B that passes the checker can be claimed to formally satisfy the fixed point. TimbukCEGAR implements an extension of [20] for $\mathcal{R}_{/E}$-automata, which means that the tool delivers correct answers. Our TimbukCEGAR is capable, in part, of analyzing Java programs by exploiting a elegant translation from the Java bytecode to term rewriting systems. Experiments show that TimbukCEGAR outperforms existing CEGAR-based completion algorithms by orders of magnitude.

Related Work. Regular Model Checking (RMC) was first applied to compute the set of reachable states of systems whose configurations are represented by words [18,14,22]. The approach was then extended to trees and applied to very simple case studies [4,19]. Other regular model checking works can be found in [2,3], where an abstraction of the transition relation allows to exploit well-quasi ordering for finite termination. Such techniques may introduce false positives; a CEGAR approach exists for the case of finite word [1], but not for the one of trees. Learning techniques apply to RMC [38,39] but trees have not yet been considered. We mention that our work extends equational abstractions [33,37] with counter-example detection and refinement. We mention the existence of other automata-based works that can handle a specific class of system [34]. CEGAR principles have been implemented in various tools such as ARMC [35] or SLAM [7]. Those specific tools are more efficient than our approach. On the other hand, RMC and rewriting rules offers a more general framework in where the abstraction and the refinements can be computed in a systematic manner.

Structure of the Paper. Section 2 introduces the basic definitions and concepts used in the paper. TRMC and Completion are introduced in Section 3. $\mathcal{R}_{/E}$-automata are introduced in Section 4. A new completion procedure is then defined in Section 5. Section 6 proposes a CEGAR approach for TRMC and Completion. Section 7 presents TimbukCEGAR. Section 8 concludes the paper and discusses future research. Due to space constraints proofs are reported in [10].

2 Background

In this section, we introduce some definitions and concepts that will be used throughout the rest of the paper (see also [6,21,30]). Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity function, and let $\mathcal{X}$ be a countable set of *variables*. $\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of *terms* and $\mathcal{T}(\mathcal{F})$ denotes the set of *ground terms* (terms without variables). The set of variables of a term t is denoted by $\mathcal{V}ar(t)$. A *substitution* is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$, which can be uniquely extended to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$. A *position* p for a term t is a word over $\mathbb{N}$. The empty sequence λ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by $\mathcal{P}os(t) = \{\lambda\}$ if $t \in \mathcal{X}$ and $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\lambda\} \cup \{i.p \mid 1 \leq i \leq n \text{ and } p \in \mathcal{P}os(t_i)\}$ otherwise. If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p and $t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s.

A *term rewriting system* (TRS) $\mathcal{R}$ is a set of *rewrite rules* $l \to r$, where $l,r \in \mathcal{T}(\mathcal{F},\mathcal{X})$, $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$. A rewrite rule $l \to r$ is *left-linear* (resp. right-linear) if each variable of l (resp. r) occurs only once in l. A TRS $\mathcal{R}$ is left-linear if every rewrite rule $l \to r$ of $\mathcal{R}$ is left-linear. A TRS $\mathcal{R}$ is said to be linear iff $\mathcal{R}$ is left-linear and right-linear. The TRS $\mathcal{R}$ induces a rewriting relation $\to_{\mathcal{R}}$ on terms as follows. Let $s,t \in \mathcal{T}(\mathcal{F},\mathcal{X})$ and $l \to r \in \mathcal{R}$, $s \to_{\mathcal{R}} t$ denotes that there exists a position $p \in \mathcal{P}os(s)$ and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$. The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\to^*_{\mathcal{R}}$ and $s \to^!_{\mathcal{R}} t$ denotes that $s \to^*_{\mathcal{R}} t$ and t is irreducible by $\mathcal{R}$. The set of $\mathcal{R}$-descendants of a set of ground terms I is $\mathcal{R}^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to^*_{\mathcal{R}} t\}$. An *equation set* E is a set of *equations* $l = r$, where $l,r \in \mathcal{T}(\mathcal{F},\mathcal{X})$. The relation $=_E$ is the smallest congruence such that for all substitution σ we have $l\sigma = r\sigma$. Given a TRS $\mathcal{R}$ and a set of equations E, a term $s \in \mathcal{T}(\mathcal{F})$ is rewritten modulo E into $t \in \mathcal{T}(\mathcal{F})$, denoted $s \to_{\mathcal{R}/E} t$, if there exist $s' \in \mathcal{T}(\mathcal{F})$ and $t' \in \mathcal{T}(\mathcal{F})$ such that $s =_E s' \to_{\mathcal{R}} t' =_E t$. Thus, the set of $\mathcal{R}$-descendants modulo E of a set of ground terms I is ${\mathcal{R}/_E}^*(I) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in I \text{ s.t. } s \to^*_{\mathcal{R}/E} t\}$.

Let Q be a finite set of symbols with arity 0, called *states*, such that $Q \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup Q)$ is called the set of *configurations*. A *transition* is a rewrite rule $c \to q$, where c is a configuration and q is state. A transition is *normalized* when $c = f(q_1,\ldots,q_n)$, $f \in \mathcal{F}$ is of arity n, and $q_1,\ldots,q_n \in Q$. A *ε-transition* is a transition of the form $q \to q'$ where q and q' are states. A bottom-up nondeterministic finite tree automaton (tree automaton for short) over the alphabet $\mathcal{F}$ is a tuple $A = \langle \mathcal{F}, Q, Q_F, \Delta \rangle$, where $Q_F \subseteq Q$, Δ is a set of normalized transitions and ε-transitions. The transitive and reflexive *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup Q)$ induced by all the transitions of A is denoted by $\to^*_A$. The tree language recognized by A in a state q is $\mathcal{L}(A,q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to^*_A q\}$. We define $\mathcal{L}(A) = \bigcup_{q \in Q_F} \mathcal{L}(A,q)$.

3 Tree Regular Model Checking with Completion

We first introduce *Tree Regular Model Checking* (TRMC), a tree automata based framework to represent possibly infinite-state systems. In TRMC, a program is

represented by a tuple $(\mathcal{F}, A, \mathcal{R})$, where $\mathcal{F}$ is an alphabet on which a set of terms $\mathcal{T}(\mathcal{F})$ can be defined; A is the tree automaton representing a possibly infinite set of configurations I, and $\mathcal{R}$ is a set of term rewriting rules that represent a transition relation *Rel*. We consider the following problem.

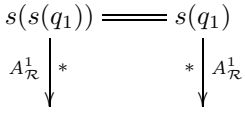
Definition 1 (Reachability Problem (RP)). *Consider a program $(\mathcal{F}, A, \mathcal{R})$ and a set of bad terms Bad. The Reachability Problem consists in checking whether there exists a term of $\mathcal{R}^*(\mathcal{L}(A))$ that belongs to Bad.*

For finite-state systems, computing the set of reachable terms $(\mathcal{R}^*(\mathcal{L}(A)))$ reduces to enumerating the terms that can be reached from the initial set of configurations. For infinite-state systems, acceleration-based methods are needed to perform this possibly infinite enumeration in a finite time. In general, such accelerations are not precise and the best one can obtain is an R-closed approximation $A^*_{\mathcal{R}}$. A tree automaton $A^*_{\mathcal{R}}$ is $\mathcal{R}$-closed if for all terms $s, t \in \mathcal{T}(\mathcal{F})$ such that $s \to_{\mathcal{R}} t$ and s is recognized by $A^*_{\mathcal{R}}$ into state q then so is t. It is easy to see that if $A^*_{\mathcal{R}}$ is $\mathcal{R}$-closed and $\mathcal{L}(A^*_{\mathcal{R}}) \supseteq \mathcal{L}(A)$, then $\mathcal{L}(A^*_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. A wide range of acceleration techniques have been developed, most of them have been discussed in Section 1. Here, we focus on Completion [24], whose objective is to computes successive automata $A^0_{\mathcal{R}} = A, A^1_{\mathcal{R}}, A^2_{\mathcal{R}}, \ldots$ that represent the effect of applying the set of rewriting rules to the initial automaton. To compute infinite sets in a finite time, each completion step is followed by a widening operator. More precisely, each application of $\mathcal{R}$, which is called a *completion step*, consists in searching for *critical pairs* $\langle t, q\rangle$ with $s \to_{\mathcal{R}} t$, $s \to^*_A q$ and $t \not\to^*_A q$. The idea being that the algorithm solves the critical pair by building from $A^i_{\mathcal{R}}$, a new tree automaton $A^{i+1}_{\mathcal{R}}$ with the additional transitions that represent the effect of applying $\mathcal{R}$. As the language recognized by A may be infinite, it is not possible to find all the critical pairs by enumerating the terms that it recognizes. The solution that was promoted in [24] consists in applying sets of substitutions $\sigma : \mathcal{X} \mapsto Q$ mapping variables of rewrite rules to states that represent infinite sets of (recognized) terms. Given a tree automaton $A^i_{\mathcal{R}}$ and a rewrite rule $l \to r \in \mathcal{R}$, to find all the critical pairs of $l \to r$ on $A^i_{\mathcal{R}}$, completion uses a *matching algorithm* [23] that produces the set of substitutions $\sigma : \mathcal{X} \mapsto Q$ and states $q \in Q$ such that $l\sigma \to^*_{A^i_{\mathcal{R}}} q$ and $r\sigma \not\to^*_{A^i_{\mathcal{R}}} q$. Solving critical pairs thus consists in adding new transitions: $r\sigma \to q'$ and $q' \to q$. Those new transitions may have to be *normalized* in order to satisfy the definition of transitions of tree automata (see [23] for details). As it was shown in [24], this operation may add not only new transitions but also new states to the automaton. In the rest of the paper, the completion-step operation will be represented by $\mathtt{C}$, i.e., the automaton obtained by applying the completion step to $A^i_{\mathcal{R}}$ is denoted $\mathtt{C}(A^i_{\mathcal{R}})$. Observe that when considering right-linear rewriting rules, we have that $\mathtt{C}$ is precise, i.e. it does not introduce in $A^{i+1}_{\mathcal{R}}$ terms that cannot be obtain from $A^i_{\mathcal{R}}$ by applying the set of rewriting rules. Observe also that if the system is non left-linear, then completion step may not produce all the reachable terms. Non left-linear rules will not be considered in the present paper.

The problem is that, except for specific classes of systems [23,25], the automaton representing the set of reachable terms cannot be obtained by applying a

finite number of completion steps. The computation process thus needs to be accelerated. For doing so, we apply a *widening operator* $\mathtt{W}$ that uses a set E of equations[1] to merge states and produce an $\mathcal{R}$-closed automaton that is an over-approximation of the set of reachable terms, i.e., an automaton $A^*_{\mathcal{R},E}$ such that $\mathcal{L}(A^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. An equation $u = v$ is applied to a tree automaton A as follows: for all substitution $\sigma : \mathcal{X} \mapsto Q$ and distinct states q_1 and q_2 such that $u\sigma \rightarrow^*_A q_1$ and $v\sigma \rightarrow^*_A q_2$, states q_1 and q_2 are merged. Completion and widening steps are applied, i.e., $A^{i+1}_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^i_{\mathcal{R},E}))$, until a $\mathcal{R}$-closed fixpoint $A^*_{\mathcal{R},E}$ is found. Our approximation framework and methodology are close to the equational abstractions of [33]. In [27], it has been shown that, under some assumptions, the widening operator may be exact, i.e., does not add terms that are not reachable.

Example 1. Let $\mathcal{R} = \{f(x) \rightarrow f(s(s(x)))\}$ be a rewriting system, $E = \{s(s(x)) = s(x)\}$ be an equation, and $A = \langle \mathcal{F}, Q, Q_F, \Delta \rangle$ be a tree automaton with $Q_F = \{q_0\}$ and $\Delta = \{a \rightarrow q_1, f(q_1) \rightarrow q_0\}$, i.e. $\mathcal{L}(A) = \{f(a)\}$. The first completion step finds the following critical pair: $f(q_1) \rightarrow^*_A q_0$ and $f(s(s(q_1))) \not\rightarrow^*_A q_0$. Hence, the completion algorithm produces $A^1_{\mathcal{R}} = \mathtt{C}(A)$ having all transitions of A plus $\{s(q_1) \rightarrow q_2, s(q_2) \rightarrow q_3, f(q_3) \rightarrow q_4, q_4 \rightarrow q_0\}$ where q_2, q_3, q_4 are new states produced by normalization of $f(s(s(q_1))) \rightarrow q_0$. Applying $\mathtt{W}$ with the equation $s(s(x)) = s(x)$ on $A^1_{\mathcal{R}}$ is equivalent to rename q_3 into q_2. The set of transitions of $A^1_{\mathcal{R},E}$ is thus $\Delta \cup \{s(q_1) \rightarrow q_2, s(q_2) \rightarrow q_2, f(q_2) \rightarrow q_4, q_4 \rightarrow q_0\}$. Completion stops on $A^1_{\mathcal{R},E}$ that is $\mathcal{R}$-closed, and thus $A^*_{\mathcal{R},E} = A^1_{\mathcal{R},E}$.



Observe that if the intersection between $A^*_{\mathcal{R},E}$ and Bad is not empty, then it does not necessarily mean that the system does not satisfy the property. Consider a set $Bad = \{f(s(a)), f(s(s(a)))\}$, the first term of this set is not reachable from A, but the second is. There is thus the need to successively refine the $\mathcal{R}$-closed automaton. The latter can be done by using a CounterExample Guided Abstraction Refinement algorithm (CEGAR). Developing such an algorithm for completion and equational abstraction is the objective of this paper.

4 $\mathcal{R}_{/E}$-Automata

Existing CEGAR approaches [17,15,16,11] check for spurious counter examples by performing a sequence of applications of the rewriting rules to $A^*_{\mathcal{R},E}$. To avoid this potentially costly step, we suggest to replace the merging of states by the addition of new rewriting rules giving information on the merging through equations. Formally:

Definition 2 ($\mathcal{R}_{/E}$-automaton). *Given a TRS $\mathcal{R}$ and a set E of equations, an $\mathcal{R}_{/E}$-automaton A is a tuple $\langle \mathcal{F}, Q, Q_F, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E \rangle$. Δ is a set of normalized*

[1] Those equations have to be provided by the user. In many cases, they can be produced when formalizing the problem in the TRMC framework [37]. The situation is similar for the predicates used in [17,15,16].

transitions. ε_E *is a set of* ε*-transitions.* $\varepsilon_{\mathcal{R}}$ *is a set of* ε*-transitions labeled by* $\top$ *or conjunctions over predicates of the form* $Eq(q, q')$ *where* $q, q' \in Q$*, and* $q \to q' \in \varepsilon_E$*.*

Set $\varepsilon_{\mathcal{R}}$ is used to distinguish a term from its successors that has been obtained by applying one or several rewriting rules. Instead of merging states according to the set of equations, A links them with epsilon transitions in ε_E. During the completion step, when exploiting critical pairs, the combination of transitions in ε_E generates transition in $\varepsilon_{\mathcal{R}}$ that are labeled with a conjunction of equations representing those transitions in ε_E. In what follows, we use $\to^*_{\Delta}$ to denote the transitive and reflexive closure of Δ. Given a set Δ of normalized transitions, the set of representatives of a state q is defined by $Rep(q) = \{t \in \mathcal{T}(\mathcal{F}) | t \to^*_{\Delta} q\}$.

Definition 3 (Run of a $\mathcal{R}_{/E}$-automaton A).

- $t|_p = f(q_1, \ldots, q_n)$ *and* $f(q_1, \ldots, q_n) \to q \in \Delta$ *then* $t \xrightarrow{\top}_A t[q]_p$
- $t|_p = q$ *and* $q \to q' \in \varepsilon_E$ *then* $t \xrightarrow{Eq(q,q')}_A t[q']_p$
- $t|_p = q$ *and* $q \xrightarrow{\alpha} q' \in \varepsilon_{\mathcal{R}}$ *then* $t \xrightarrow{\alpha}_A t[q']_p$
- $u \xrightarrow{\alpha}_A v$ *and* $v \xrightarrow{\alpha'}_A w$ *then* $u \xrightarrow{\alpha \wedge \alpha'}_A w$

Theorem 1. $\forall t \in \mathcal{T}(\mathcal{F} \cup Q),\ q \in Q,\ t \xrightarrow{\alpha}_A q \iff t \to^*_A q$

A run $\xrightarrow{\alpha}$ abstracts a rewriting path of $\to_{\mathcal{R}/E}$. If $t \xrightarrow{\alpha} q$, then there exists a term $s \in Rep(q)$ such that $s \to^*_{\mathcal{R}/E} t$. The formula α denotes the subset of transitions of ε_E needed to recognize t into q.

Example 2. Let $I = f(a)$ be an initial set of terms, $\mathcal{R} = \{f(c) \to g(c),\ a \to b\}$ be a set of rewriting rules, and $E = \{b = c\}$ be a set of equations. We build A an overapproximation automaton for $\mathcal{R}^*(I)$, using E.

Thanks to ε-transitions, the automaton A represented in Fig. 1 contains some information about the path used to reach terms using $\mathcal{R}$ and E. Each state has a representative term from which others are obtained. The equality $b = c$ is represented by the two transitions $q_c \to q_b$ and $q_b \to q_c$ of ε_E, taking into account that b and c are the representatives terms for states q_b and q_c, respectively. Consider now State q_c, Transition $q_b \to q_c$ indicates that the term b is obtained from Term c by using the equality. Conversely, Transition $q_c \to q_b$ leads to the conclusion that Term c is obtained from Term b.

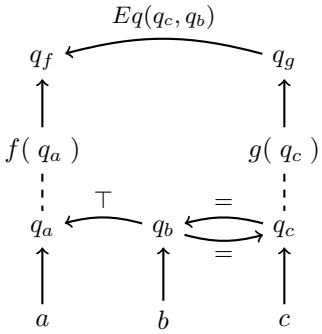


Fig. 1. Automaton A

The transition $q_b \to q_a$ denotes that the term b is a descendant of a by rewriting. Using Definition 3, the runs $f(c) \xrightarrow{Eq(q_c,q_b)} q_f$ indicates that to obtain $f(c)$ from $f(a)$ – the representative term of q_f – we used the equality $b = c$, which is obtained from $q_c \to q_b$. We indeed observe $f(a) \to_{\mathcal{R}} f(b) =_E f(c)$. If we now consider the transition $q_g \to q_f$ we labeled the transition with the formula $Eq(q_c, q_b)$. To reach $g(c)$ from $f(a)$, we rewrite $f(c)$. We have seen this term is reachable thanks to the equivalence relation induced by $b = c$. By transitivity, this equivalence is also used to reach the term $g(c)$. We thus label the transition of $\varepsilon_{\mathcal{R}}$ to save this information. We obtain the run $g(c) \xrightarrow{Eq(q_c,q_b)} q_f$. We observe that the transition $q_b \to q_a$ is labeled by the formula $\top$ since b is reachable from a without any equivalence. By congruence, so is $f(b)$ from $f(a)$. The run $f(b) \xrightarrow{\top} q_f$ denotes it.

We now introduce a property that will be used in the refinement procedure to distinguish between counter-examples and false positives.

Definition 4 (A well-defined $\mathcal{R}_{/E}$-automaton). *A is a* well-defined $\mathcal{R}_{/E}$*-automaton, if :*

- *For all states q of A, and all terms v such that $v \xrightarrow{\top}_A q$, there exists u a term representative of q such that $u \to^*_{\mathcal{R}} v$*
- *If $q \xrightarrow{\phi} q'$ is a transition of $\varepsilon_{\mathcal{R}}$, then there exist terms $s, t \in \mathcal{T}(\mathcal{F})$ such that $s \xrightarrow{\phi}_A q$, $t \xrightarrow{\top}_A q'$ and $t \to_{\mathcal{R}} s$.*

The first item in Definition 4 guarantees that every term recognized by using transitions labeled with the formula $\top$ is indeed reachable from the initial set. The second item is used to refine the automaton. A rewriting step of $\to_{\mathcal{R}/E}$ denoted by $q \xrightarrow{\phi} q'$ holds thanks to some transitions of ε_E that occurs in ϕ. If we remove transitions in ε_E in such a way that ϕ does not hold, then the transition $q \xrightarrow{\phi} q'$ should also be removed.

According to the above construction, a term t that is recognized by using at least a transition labeled with a formula different from $\top$ can be removed from the language of the $\mathcal{R}_{/E}$-automaton by removing some transitions in ε_E. This "pruning" operation will be detailed in Section 6.

5 Solving the Reachability Problem with $\mathcal{R}_{/E}$-Automaton

In this section, we extend the completion and widening principles introduced in Section 3 to take advantage of the structure of $\mathcal{R}_{/E}-$automata. We consider an initial set I that can be represented by a tree automaton $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q^0, Q_F, \Delta^0 \rangle$, and transition relation represented by a set of linear rewriting rules $\mathcal{R}$. In the next section, we will see that the right-linearity condition may be relaxed using additionnal hypotheses. We compute successive approximations $A^i_{\mathcal{R},E} = \langle \mathcal{F}, Q^i, Q_f, \Delta^i \cup \varepsilon^i_{\mathcal{R}} \cup \varepsilon^i_E \rangle$ from $A^0_{\mathcal{R},E}$ using $A^{i+1}_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^i_{\mathcal{R},E}))$. Observe that $A^0_{\mathcal{R},E}$ is well-defined as the sets $\varepsilon^0_{\mathcal{R}}$ and ε^0_E are empty.

5.1 The Completion Step C

Extending completion to $\mathcal{R}_{/E}$-automaton requires to modify the concept of critical pair and so the algorithm to compute them. A critical pair for a $\mathcal{R}_{/E}$-automaton is a triple $\langle r\sigma, \alpha, q\rangle$ such that $l\sigma \to r\sigma$, $l\sigma \xrightarrow{\alpha}_{A^i_{\mathcal{R},E}} q$ and there is no formula α' such that $r\sigma \xrightarrow{\alpha'}_{A^i_{\mathcal{R},E}} q$. The resolution of such a critical pair consists of adding to $\mathtt{C}(A^i_{\mathcal{R},E})$ the transitions to obtain $r\sigma \xrightarrow{\alpha}_{\mathtt{C}(A^i_{\mathcal{R},E})} q$. This is followed by a normalization step `Norm` whose definition is similar to the one for classical tree automata.

Definition 5 (Normalization). *The normalization is done in two mutually inductive steps parametrized by the configuration c to recognize, and by the set of transitions Δ to extend. Let Q^{Δ}_{new} be a set of (new) states not occurring in Δ.*

$$\begin{cases} \mathtt{Norm}(c,\Delta) = \mathtt{Slice}(d,\Delta), & \text{for one } d \text{ s.t. } c \to^*_{\Delta} d, \text{ with } c,\ d \in \mathcal{T}(\mathcal{F}\cup Q) \\ \mathtt{Slice}(q,\Delta) = \Delta, & q \in Q \\ \mathtt{Slice}(f(q_1,\ldots,q_n),\Delta) = \Delta \cup \{f(q_1,\ldots,q_n)\to q\}, & q_i \in Q \text{ and one } q \in Q^{\Delta}_{new} \\ \mathtt{Slice}(f(t_1,\ldots,t_n),\Delta) = \mathtt{Norm}(f(t_1,\ldots,t_n),\mathtt{Slice}(t_i,\Delta)), & \exists t_i \in \mathcal{T}(\mathcal{F}\cup Q)\setminus Q \end{cases}$$

Definition 6 (Resolution of a critical pair). *Given a $\mathcal{R}_{/E}$-automaton $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E\rangle$ and a critical pair $p = \langle r\sigma, \alpha, q\rangle$, the* resolution *of p on A is the $\mathcal{R}_{/E}$-automaton $A' = \langle \mathcal{F}, Q', Q_f, \Delta' \cup \varepsilon'_{\mathcal{R}} \cup \varepsilon_E\rangle$ where*

- $\Delta' = \Delta \cup \mathtt{Norm}(r\sigma, \Delta \setminus \Delta^0)$;
- $\varepsilon'_{\mathcal{R}} = \varepsilon_{\mathcal{R}} \cup \{q' \xrightarrow{\alpha} q\}$ *where q' is the state such that $r\sigma \to_{\Delta'\setminus\Delta_0} q'$;*
- *Q' is the union of Q with the set of states added when creating Δ'.*

Note that Δ_0, the set of transitions of $A^0_{\mathcal{R}}$, is not used in the normalization process. This is to guarantee that A' is well-defined. The $\mathcal{R}_{/E}$-automaton $\mathtt{C}(A^i_{\mathcal{R},E})$ is obtained by recursively applying the above resolution principle to all critical pairs p of the set of critical pairs between $\mathcal{R}$ and $A^i_{\mathcal{R},E}$.

The set of all critical pairs is obtained by solving the *matching problems* $l \trianglelefteq q$ for all rewrite rules $l \to r \in \mathcal{R}$ and all states $q \in A^i_{\mathcal{R},E}$. Solving $l \trianglelefteq q$ is performed in two steps. First, one computes S, that is the set of all pairs (α, σ) such that α is a formula, σ is a substitution of $\mathcal{X} \mapsto Q^i$, and $l\sigma \xrightarrow{\alpha} q$. The formula α is a conjunction of Predicates $Eq(q', q'')$ that denotes the used transitions of ε_E to rewrite $l\sigma$ in q, in accordance with Definition 3. Due to space constraints the algorithm, which always terminates, can be found in [10].

Second, after having computed S for $l \trianglelefteq q$, we identify elements of the set that correspond to critical pairs. By definition of S, we know that there exists a transition $l\sigma \xrightarrow{\alpha}_{A^i_{\mathcal{R},E}} q$ for $(\alpha,\sigma) \in S$. If there exists a transition $r\sigma \xrightarrow{\alpha'}_{A^i_{\mathcal{R},E}} q$, then $r\sigma$ has already been added to $A^i_{\mathcal{R},E}$. If there does not exist a transition of the form $r\sigma \xrightarrow{\alpha'}_{A^i_{\mathcal{R},E}} q$, then $\langle r\sigma, \alpha', q\rangle$ is a critical pair to solve on $A^i_{\mathcal{R},E}$. The following theorem shows that our methodology is complete.

Theorem 2. *If $A^i_{\mathcal{R},E}$ is well-defined then so is $\mathtt{C}(A^i_{\mathcal{R},E})$, and $\forall q \in Q^i$, $\forall t \in \mathcal{L}(A^i_{\mathcal{R},E}, q)$, $\forall t' \in \mathcal{T}(\mathcal{F})$, $t \to_{\mathcal{R}} t' \implies t' \in \mathcal{L}(\mathtt{C}(A^i_{\mathcal{R},E}), q)$.*

Example 3. Let $\mathcal{R} = \{f(x) \to f(s(s(x)))\}$ be a set of rewriting rules and $A^0_{\mathcal{R},E} = \langle \mathcal{F}, Q, Q_F, \Delta^0 \rangle$ be a tree automaton such that $Q_F = \{q_0\}$ and $\Delta^0 = \{a \to q_1, f(q_1) \to q_0\}$. The solution of the matching problem $f(x) \trianglelefteq q_0$ is $S = \{(\sigma, \phi)\}$, with $\sigma = \{x \to q_1\}$ and $\phi = \top$. Hence, since $f(s(s(q_1))) \xrightarrow{\top}_{A^0_{\mathcal{R},E}} q_0$, $\langle f(s(s(q_1))), \top, q_0 \rangle$ is the only critical pair to be solved. So, we have $\mathtt{C}(A^0_{\mathcal{R},E}) = \langle \mathcal{F}, Q^1, Q_F, \Delta^1 \cup \varepsilon^1_{\mathcal{R}} \cup \varepsilon^0_E \rangle$, with:

$$\Delta^1 = \mathtt{Norm}(f(s(s(q_1))), \emptyset) \cup \Delta^0 = \{s(q_1) \to q_2, s(q_2) \to q_3, f(q_3) \to q_4\} \cup \Delta^0,$$
$$\varepsilon^1_{\mathcal{R}} = \{q_4 \xrightarrow{\top} q_0\}, \text{since } f(s(s(q_1))) \to_{\Delta^1 \setminus \Delta^0} q_4, \varepsilon^0_E = \emptyset \text{ and } Q^1 = \{q_0, q_1, q_2, q_3, q_4\}.$$

Observe that if $\mathtt{C}(A^i_{\mathcal{R},E}) = A^i_{\mathcal{R},E}$, then we have reached a fixpoint.

5.2 The Widening Step W

Consider a $\mathcal{R}_{/E}$-automaton $A = \langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E \rangle$, the widening consists in computing a $\mathcal{R}_{/E}$-automaton $\mathtt{W}(A)$ that is obtained from A by using E.

For each equation $l = r$ in E, we consider all pair (q, q') of distinct states of Q^i such that there exists a substitution σ to obtain the following diagram. Observe that $\xrightarrow{=}_A$, the transitive and reflexive rewriting relation induced by $\Delta \cup \varepsilon_E$, defines particular runs which exclude transitions of $\varepsilon_{\mathcal{R}}$. This allows us to build a more accurate approximation. The improvement in accurary is detailed in [27].

$$\begin{array}{ccc} l\sigma & \overset{E}{=\!=} & r\sigma \\ {\scriptstyle A}\downarrow{\scriptstyle =} & & {\scriptstyle A}\downarrow{\scriptstyle =} \\ q & & q' \end{array}$$

Intuitively, if we have $u \xrightarrow{=}_A q$, then we know that there exists a term t of $Rep(q)$ such that $t =_E u$. The automaton $\mathtt{W}(A)$ is given by the tuple $\langle \mathcal{F}, Q, Q_f, \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon'_E \rangle$, where ε'_E is obtained by adding the transitions $q \to q'$ and $q' \to q$ to ε_E (for each pair (q, q')).

Theorem 3. *Assuming that A is well-defined, we have A syntactically included in $\mathtt{W}(A)$, and $\mathtt{W}(A)$ is well-defined.*

Example 4. Consider the $\mathcal{R}_{/E}$-automaton $\mathtt{C}(A^0_{\mathcal{R},E})$ given in Example 3. Using Equation $s(s(x)) = s(x)$, we compute $A^1_{\mathcal{R},E} = \mathtt{W}(\mathtt{C}(A^0_{\mathcal{R},E}))$. We have $\sigma = \{x \mapsto q_1\}$ and the following diagram. We then obtain $A^1_{\mathcal{R},E} = \langle \mathcal{F}, Q^1, Q_f, \Delta^1 \cup \varepsilon^1_{\mathcal{R}} \cup \varepsilon^1_E \rangle$, where $\varepsilon^1_E = \varepsilon^0_E \cup \{q_3 \to q_2,\ q_2 \to q_3\}$ and $\varepsilon^0_E = \emptyset$. Observe that $A^1_{\mathcal{R},E}$ is a fixpoint, i.e., $\mathtt{C}(A^1_{\mathcal{R},E}) = A^1_{\mathcal{R},E}$.

$$\begin{array}{ccc} s(s(q_1)) & \overset{E}{=\!=} & s(q_1) \\ {\scriptstyle \mathtt{C}(A^0_{\mathcal{R},E})}\downarrow{\scriptstyle =} & & {\scriptstyle \mathtt{C}(A^0_{\mathcal{R},E})}\downarrow{\scriptstyle =} \\ q_3 & & q_2 \end{array}$$

6 A CEGAR Procedure for $\mathcal{R}_{/E}$-Automata

Let $\mathcal{R}$ be a TRS, I be a set of initial terms characterized by the $\mathcal{R}_{/E}-$automaton $A^0_{\mathcal{R},E}$ and Bad the set of forbidden terms represented by A_{Bad}. We now complete our CEGAR approach by proposing a technique that checks whether a term is indeed reachable from the initial set of terms. If the term is a spurious counter-example i.e. a counter-example of the approximation, then it has to be removed

from the approximation automatically, else one can deduce that the involved term is actually reachable.

Let $A^k_{\mathcal{R},E} = \langle \mathcal{F}, Q^k, Q_f, \Delta^k \cup \varepsilon^k_{\mathcal{R}} \cup \varepsilon^k_E \rangle$ be a $\mathcal{R}_{/E}$-automaton obtained after k steps of completion and widening from $A^0_{\mathcal{R},E}$ and assume that $\mathcal{L}(A^k_{\mathcal{R},E}) \cap Bad \neq \emptyset$. Let $S_{A^k_{\mathcal{R},E} \cap A_{Bad}}$ be a set of triples $\langle q, q', \phi \rangle$ where q is a final state of $A^k_{\mathcal{R},E}$, q' is a final state of A_{Bad} and ϕ is a formula on transitions of ε^k_E and such that for each triple (q, q', ϕ), the formula ϕ holds if and only if there exists $t \in \mathcal{L}(A^k_{\mathcal{R},E}, q) \cap \mathcal{L}(A_{Bad}, q')$ and $t \xrightarrow{\phi}_{A^k_{\mathcal{R},E}} q$. Note that $S_{A^k_{\mathcal{R},E} \cap A_{Bad}}$ can be obtained using an intersection based algorithm defined in [10]. We consider two cases. First, as $A^k_{\mathcal{R},E}$ is well-defined, if $\phi = \top$, we deduce that t is indeed a reachable term. Otherwise, ϕ is a formula whose atoms are of the form $Eq(q_j, q'_j)$, and t is possibly a spurious counter-example, and the run $t \xrightarrow{\phi}_{A^k_{\mathcal{R},E}} q$ must be removed. Refinement consists in computing a pruned version $\mathtt{P}(A^k_{\mathcal{R},E}, S_{A^k_{\mathcal{R},E} \cap A_{Bad}})$ of $A^k_{\mathcal{R},E}$.

Definition 7. *Given an $\mathcal{R}_{/E}-$automaton $A = \langle \mathcal{F}, Q, Q_F, \Delta_0 \cup \Delta \cup \varepsilon_{\mathcal{R}} \cup \varepsilon_E \rangle$ and a set of terms specified by the automaton A_{Bad}, the prune process is defined by*

$$\mathtt{P}(A, S_{A \cap A_{Bad}}) = \begin{cases} \mathtt{P}(A', S_{A' \cap A_{Bad}}) & \text{if } S_{A' \cap A_{Bad}} \neq \emptyset \text{ and with} \\ & A' = \mathtt{Clean}(A, S_{A \cap A_{Bad}}) \\ A & \text{if } S_{A \cap A_{Bad}} = \emptyset \text{ or there exists } t \in Bad \\ & \text{s.t. } t \xrightarrow{\top}_A q_f \text{ and } q_f \in Q_F. \end{cases}$$

where $\mathtt{Clean}(A, S_{A \cap A_{Bad}})$, consists of removing transitions of ε_E until for each $\langle q_f, q'_f, \phi \rangle \in S_{A \cap A_{Bad}}$, ϕ does not hold, i.e., $\phi = \bot$ with q_f, q'_f respectively two final states of A and A_{Bad}.

To replace Predicate $Eq(q, q')$ by $\bot$ in ϕ, we have to remove the transition $q \to q'$ from ε_E. In addition, we also have to remove all transitions $q \xrightarrow{\alpha} q' \in \varepsilon_{\mathcal{R}}$, where the conjunction α contains some predicates $Eq(q_1, q_2)$ whose transition $q_1 \to q_2$ has been removed from ε_E. In general, removing Transition $q \to q'$ may be too rough. Indeed, assuming that there also exists a transition $q'' \to q$ of ε_E, removing the transition $q \to q'$ also avoids the induced reduction $q'' \to q'$ from the automaton and then, unconcerned terms of q'' are also removed. To save those terms, Transition $q'' \to q'$ is added to ε_E, but only if it has never been removed by a pruning step. This point is important to refine the automaton with accuracy. The prune step is called recursively as inferred transitions may keep the intersection non-empty.

Theorem 4. *Let $t \in Bad$ be a spurious counter-example. The pruning process always terminates, and removes all the runs of the form $t \xrightarrow{\phi} q$.*

Example 5. We consider the $\mathcal{R}_{/E}$-automaton A of Example 2. It is easy to see that A recognizes the term $g(c)$. Indeed, by Definition 3, we have $g(c) \xrightarrow{Eq(q_c,q_b)} q_f$. Consider now the rewriting path $f(a) \rightarrow_{\mathcal{R}} f(b) =_E f(c) \rightarrow_{\mathcal{R}} g(c)$. If we remove the step $f(b) =_E f(c)$ denoted by the transition $q_c \rightarrow q_b$, then $g(c)$ becomes unreachable and should also be removed. The first step in pruning A consists thus in removing this transition. In a second step, we propagate the information by removing all transition of $\varepsilon_{\mathcal{R}}$ labeled by a formula that contains $Eq(q_c, q_b)$. This is done to remove all terms obtained by rewriting with the equivalence $b =_E c$. After having pruned all the transitions, we observe that the terms recognized by A are given by the set $\{f(a), f(b)\}$.

Let us now characterize the soundness and completness of our approach.

Theorem 5 (Soundness on left-linear TRS). *Consider a left-linear TRS $\mathcal{R}$, a set of terms Bad, a set of equations E and a well-defined $\mathcal{R}_{/E}-$automaton A_0. Let $A^*_{\mathcal{R},E}$ be a fixpoint $\mathcal{R}_{/E}$-automaton of $\mathtt{P}(A', S_{A' \cap A_{Bad}})$ and $A' = \mathtt{W}(\mathtt{C}(A_i))$ for $i \geq 0$. If $\mathcal{L}(A^*_{\mathcal{R},E}) \cap Bad = \emptyset$, then $Bad \cap \mathcal{R}^*(\mathcal{L}(A_0)) = \emptyset$.*

Theorem 6 (Completeness on Linear TRS). *Given a linear TRS $\mathcal{R}$, a set of terms Bad defined by automata A_{Bad}, a set of equations E and a well-defined $\mathcal{R}_{/E}-$automaton A_0. For any $i > 0$, let A_i be the $\mathcal{R}_{/E}-$automaton obtained from A_{i-1} in such a way: $A_i = \mathtt{P}(A', S_{A' \cap A_{Bad}})$ and $A' = \mathtt{W}(\mathtt{C}(A_{i-1}))$. If $Bad \cap \mathcal{R}^*(\mathcal{L}(A_0)) \neq \emptyset$ then there exists $t \in Bad$ and $j > 0$ such that $t \xrightarrow{\top}_{A_j} q_f$ and q_f is a final state of A_j.*

This result also extends to left-linear TRS with a finite set of initial terms (cardinality of $Rep(q)$ is 1 for all state q of A_0).

Theorem 7 (Completeness on Left-Linear TRS). *Theorem 6 extends to left-linear TRS if for any state q of A_0, the cardinality of $Rep(q)$ is 1.*

7 Implementation, Application and Certification

Our approach has been implemented in TimbukCEGAR that is an extension of the Timbuk 3.1 toolset [29]. Timbuk is a well-acknowledged tree automata library that implements several variants of the completion approach. TimbukCEGAR consists of around 11000 lines of OCaml, 75% of which are common with Timbuk 3.1. TimbukCEGAR exploits a BDD-based representation of equation formulas through the Buddy BDD library [32].

A particularity of TimbukCEGAR is that it is certified. At the heart of any abstraction algorithm there is the need to check whether a candidate over-approximation B is indeed a fixed point, that is if $\mathcal{L}(B) \supseteq \mathcal{R}^*(\mathcal{L}(A))$. Such check has been implemented in various TRMC toolsets, but there is no guarantee that it behaves correctly, i.e., that the TRMC toolset gives a correct answer. In [20], a checker for tree automata completion was designed and proved correct using the Coq [9] proof assistant. As such, any TRMC toolset that produces an automaton B that passes the checker can be claimed to work properly. TimbukCEGAR

implements a straightforward extension of [20] for $\mathcal{R}_{/E}$-automata, which means that the tool delivers provably correct answers. In what follows, we describe how Java programs can be analyzed using our approach. Both Timbuk and TimbukCEGAR are available at `http://www.irisa.fr/celtique/genet/timbuk/`.

In a french initiative called RAVAJ [36], we have defined a generic certified verification chain based on TRMC. This chain is composed of three main links. The two first links rely on an encoding of the operational semantics of the programming language as a term rewriting system and a set of rewrite rules. The third link is a TRMC toolset, here TimbukCEGAR. With regards to classical static analysis, the objective is to use TRMC and particularly tree automata completion as a foundation mechanism for ensuring, by construction, safety of static analyzers. For Java, using approximation rules instead of abstract domains makes the analysis easier to fine-tune. Moreover, our approach relies on a checker that certifies the answer to be correct.

We now give more details and report some experimental results. We used Copster [8], to compile a Java `.class` file into a TRS. The obtained TRS models exactly a subset of the semantics[2] of the Java Virtual Machine (JVM) by rewriting a term representing the state of the JVM [13]. States are of the form `IO(st,in,out)` where `st` is a program state, `in` is an input stream and `out` an output stream. A program state is a term of the form `state(f,fs,h,k)` where `f` is current frame, `fs` is the stack of calling frames, `h` a heap and `k` a static heap. A frame is a term of the form `frame(m,pc,s,l)` where `m` is a fully qualified method name, `pc` a program counter, `s` an operand stack and `t` an array of local variables. The frame stack is the call stack of the frame currently being executed: `f`. We consider the following program:

```
class List{
    List next;
    int val;
    public List(int elt, List l){
        next= l;
        if (elt<0) val= -elt;
        else val= elt;
    }
    public void printSum(){
        List l= this;
        int sum= 0;
        while (l != null){
            sum= sum+l.val;
            l= l.next;
        }
        System.out.println(sum);
    }}
```

```
class TestList{
    public static void main(String[] argv){
        List ls= null;
        int x= 0;
        while (x!=-1) {
            try {x= System.in.read();}
            catch(java.io.IOException e){};
            ls= new List(x,ls);
        }
        ls.printSum();
    }
}
```

Let us now check that the sum output by the program can never be equal to zero, for all non-empty input stream of integers. The TRS generated by Copster has 879 rules encoding both the JVM semantics and the bytecode of the above Java program. Note that, this example is bigger than those generally used by other TRMC techniques. The complete TRS is available with TimbukCEGAR distribution. Initial terms are of the form `IO(s,lin,nilout)` where `s` is the

[2] Essentially basic types, arithmetic, object creation, field manipulation, virtual method invocation, as well as a subset of the String library.

initial JVM state, `lin` is a non-empty unbounded list of integers and `nilout` is the empty list of outputs. Starting from this initial set of terms, completion is likely to diverge without approximations. Indeed, the program is going to allocate infinitely many objects of class `List` in the heap and, furthermore, compute an unbounded sum in the method `printSum`. In the heap, there is one separate heap for each class. Each heap consists of a list of objects. For instance, in the heap for class `List`, objects are stored using a list constructor `stackHeapList(x,y)`. Thus, to enforce termination we can approximate the heap for objects of class `List` using the following equation `stackHeapList(x,y)=y`. The effect of this equation is to collapse all the possible lists built using `stackHeapList`, hence all the possible heaps for class `List`. The other equations are `succ(x)=x` and `pred(x)=x` for approximating infinitely growing or decreasing integers.

By using those equations, TimbukCEGAR finds a counterexample. This is due to the fact that, amongst all considered input streams, an input stream consisting of a list of 0 results into a 0 sum. The solution is to restrict the initial language to non-empty non-zero integer streams. However, refinement of equations is needed since `succ(x)=x` and `pred(x)=x` put 0 and all the other integers in the same equivalence class. Refining those equations by hand is hard, *e.g.* using equations `succ(succ(x))=succ(x)` and `pred(pred(x))=pred(x)` is not enough to eliminate spurious counterexamples. After 334 completion steps and 4 refinement steps, TimbukCEGAR is able to complete the automaton and achieve the certified proof. The resulting automaton produced by the tool has 3688 transitions which are produced in $128s$. Then, it can be Coq-certified in $17017s$. The memory usage for the whole process does not exceed $531Mb$. One of the reasons for which certifying automata produced by TimbukCEGAR takes more time than for Timbuk 3.1 is that the checker has to normalize epsilon transitions of $\mathcal{R}_{/E}$-automata. This is straightforward but may cause an explosion of the size of the tree automaton to be checked. However this can be improved a lot by defining a specific Coq-checker for $\mathcal{R}_{/E}$-automata. It is worth mentioning that the TRS produced by Copster from Java programs are left-linear but not right-linear. Hence, our soundness theorem applies but not the completeness theorem, since the TRS is not right-linear and the initial language is not finite. However, here completion steps do not introduce spurious counter examples. Some other examples of application can be found in [10]. Those experiments show, in particular, that the overhead due to the use of $\mathcal{R}_{/E}$-automata in completion can be limited thanks to BDDs. As a consequence, TimbukCEGAR performances are similar to those of Timbuk 3.1 when no refinement is performed, and TimbukCEGAR outperforms the other known CEGAR completion implementation when refinement is needed.

8 Conclusion

We have presented a new CounterExample Guided Abstraction Refinement procedure for TRMC based on equational abstraction. Our approach has been implemented in TimbukCEGAR that is the first TRMC toolset certified correct. Our approach leads, in particular, to a Java program analyzer starting from code

to verification. Unlike most of existing works, our method works with the Java semantics itself rather than with some abstract model that is statically or even manually derived from the code. One additional feature is that the complete verification is certified by an external proof assistant. We are convinced that our work open news doors in application of RMC approaches to rigorous system design. One challenge for future work is definitively to consider non left-linear TRS – a formalism that can be used to model cryptographic protocols. In [25,26,28,5], it has been shown that an extension of the completion algorithm can lead to a powerful verification toolset for cryptographic protocols. However, this existing setting is still lacking an abstraction refinement procedure to be made fully automatic. On the one side, the theoretical challenge is thus to extend the CEGAR completion to non left-linear TRS. On the other side, the technical challenge is to extend the Coq checker to handle non left-linear TRS and $\mathcal{R}_{/E}$-automata, in order to improve the Coq-checking time. Tackling those two goals will allow us to propose the *first certified automatic verification tool for security protocols*, a major advance in the formal verification area.

Acknowledgements. Thanks to F. Besson for his help in integrating Buddy.

References

1. Abdulla, P.A., Chen, Y.-F., Delzanno, G., Haziza, F., Hong, C.-D., Rezine, A.: Constrained Monotonic Abstraction: A CEGAR for Parameterized Verification. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 86–101. Springer, Heidelberg (2010)
2. Abdulla, P.A., Delzanno, G., Rezine, A.: Parameterized Verification of Infinite-State Processes with Global Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 145–157. Springer, Heidelberg (2007)
3. Abdulla, P.A., Ben Henda, N., Delzanno, G., Haziza, F., Rezine, A.: Parameterized Tree Systems. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 69–83. Springer, Heidelberg (2008)
4. Abdulla, P.A., Legay, A., d'Orso, J., Rezine, A.: Simulation-Based Iteration of Tree Transducers. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 30–44. Springer, Heidelberg (2005)
5. Avispa – a tool for Automated Validation of Internet Security Protocols, `http://www.avispa-project.org`
6. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
7. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
8. Barré, N., Besson, F., Genet, T., Hubert, L., Le Roux, L.: Copster homepage (2009), `http://www.irisa.fr/celtique/genet/copster`
9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer (2004)

10. Boichut, Y., Boyer, B., Genet, T., Legay, A.: Fast Equational Abstraction Refinement for Regular Tree Model Checking. Technical report, INRIA (2010), http://hal.inria.fr/inria-00501487
11. Boichut, Y., Courbis, R., Héam, P.-C., Kouchnarenko, O.: Finer Is Better: Abstraction Refinement for Rewriting Approximations. In: Voronkov, A. (ed.) RTA 2008. LNCS, vol. 5117, pp. 48–62. Springer, Heidelberg (2008)
12. Boichut, Y., Dao, T.-B.-H., Murat, V.: Characterizing Conclusive Approximations by Logical Formulae. In: Delzanno, G., Potapov, I. (eds.) RP 2011. LNCS, vol. 6945, pp. 72–84. Springer, Heidelberg (2011)
13. Boichut, Y., Genet, T., Jensen, T., Le Roux, L.: Rewriting Approximations for Fast Prototyping of Static Analyzers. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 48–62. Springer, Heidelberg (2007)
14. Boigelot, B., Legay, A., Wolper, P.: Iterating Transducers in the Large (Extended Abstract). In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 223–235. Springer, Heidelberg (2003)
15. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract regular tree model checking. ENTCS 149(1), 37–48 (2006)
16. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
17. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract Regular Model Checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
18. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular Model Checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
19. Bouajjani, A., Touili, T.: Extrapolating Tree Transformations. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 539–554. Springer, Heidelberg (2002)
20. Boyer, B., Genet, T., Jensen, T.: Certifying a Tree Automata Completion Checker. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 523–538. Springer, Heidelberg (2008)
21. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M.: Tree automata techniques and applications (2008)
22. Dams, D., Lakhnech, Y., Steffen, M.: Iterating transducers. Journal of Logic and Algebraic Programming (JLAP) 52-53, 109–127 (2002)
23. Feuillade, G., Genet, T., Viet Triem Tong, V.: Reachability Analysis over Term Rewriting Systems. Journal of Automated Reasonning 33(3-4), 341–383 (2004)
24. Genet, T.: Decidable Approximations of Sets of Descendants and Sets of Normal Forms. In: Nipkow, T. (ed.) RTA 1998. LNCS, vol. 1379, pp. 151–165. Springer, Heidelberg (1998)
25. Genet, T.: Reachability analysis of rewriting for software verification. Université de Rennes 1, Habilitation (2009)
26. Genet, T., Klay, F.: Rewriting for Cryptographic Protocol Verification. In: McAllester, D. (ed.) CADE 2000. LNCS (LNAI), vol. 1831, pp. 271–290. Springer, Heidelberg (2000)
27. Genet, T., Rusu, R.: Equational tree automata completion. JSC 45 (2010)
28. Genet, T., Tang-Talpin, Y.-M., Viet Triem Tong, V.: Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In: WITS 2003 (2003)

29. Genet, T., Viet Triem Tong, V.: Timbuk 2.0 – a Tree Automata Library. IRISA/Université de Rennes 1 (2001), http://www.irisa.fr/celtique/genet/timbuk/
30. Gilleron, R., Tison, S.: Regular tree languages and rewrite systems. Fundamenta Informaticae 24, 157–175 (1995)
31. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic Model Checking with Rich Assertional Languages. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 424–435. Springer, Heidelberg (1997)
32. Lind-Nielsen, J.: Buddy 2.4 (2002), http://buddy.sourceforge.net
33. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. TCS 403, 239–264 (2008)
34. Patin, G., Sighireanu, M., Touili, T.: SPADE: Verification of Multithreaded Dynamic and Recursive Programs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 254–257. Springer, Heidelberg (2007)
35. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
36. Ravaj: Rewriting and Approximations for Java Applications Verification, http://www.irisa.fr/celtique/genet/RAVAJ
37. Takai, T.: A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In: van Oostrom, V. (ed.) RTA 2004. LNCS, vol. 3091, pp. 119–133. Springer, Heidelberg (2004)
38. Vardhan, A., Sen, K., Viswanathan, M., Agha, G.: Using Language Inference to Verify Omega-Regular Properties. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 45–60. Springer, Heidelberg (2005)
39. Vardhan, A., Viswanathan, M.: LEVER: A Tool for Learning Based Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 471–474. Springer, Heidelberg (2006)

SMT-Based False Positive Elimination in Static Program Analysis

Maximilian Junker[1], Ralf Huuck[2], Ansgar Fehnker[2], and Alexander Knapp[3]

[1] Technische Universität München, Munich, Germany
junkerm@in.tum.de
[2] NICTA, University of New South Wales, Sydney, Australia
{ansgar.fehnker,ralf.huuck}@nicta.com
[3] Universität Augsburg, Augsburg, Germany
knapp@informatik.uni-augsburg.de

Abstract. Static program analysis for bug detection in large C/C++ projects typically uses a high-level abstraction of the original program under investigation. As a result, so-called false positives are often inevitable, i.e., warnings that are not true bugs. In this work we present a novel abstraction refinement approach to automatically investigate and eliminate such false positives. Central to our approach is to view static analysis as a model checking problem, to iteratively compute infeasible sub-paths of infeasible paths using SMT solvers, and to refine our models by adding observer automata to exclude such paths. Based on this new framework we present an implementation of the approach into the static analyzer Goanna and discuss a number of real-life experiments on larger C code projects, demonstrating that we were able to remove most false positives automatically.

1 Introduction

Static program analysis of industrial size C/C++ programs for the detection of quality as well as security bugs has had some considerable success in the recent years. A number of software tools and companies [23,12] resulted from theoretical advances and increased computing power, leading to the detection of complex source code defects with minimal effort from the side of the developers.

However, static analysis techniques are based on approximations of the original source code semantics. As such, the results of static analyzers might contain spurious warnings, i.e., false positives. The task of assessing the validity of tool warnings falls back to the developer. But with the increasing complexity of the bugs that those techniques can uncover, this assessment is getting more and more difficult. In large software projects developers may be forced to spend a lot of time reconstructing a warning of a static analysis tool just to discover that the claimed bug is not real. Therefore, it is vital for static analysis tools not only to find many complex bugs, but also to assure that the majority of those are not false positives.

Unlike static program analysis, traditional software model checking has established methods in dealing with abstractions and false positives, which are referred to as spurious counter-examples. One particular prominent method is counter-example guided

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 316–331, 2012.

abstraction refinement (CEGAR) [6]. In the world of static program analysis, however, there is no good notion of automatic iterative refinement. Moreover, CEGAR approaches typically refine in each iteration the whole program/function under consideration and re-run the analysis on the new model, which can often be costly.

In this work we adopt some ideas from such established techniques, but take a significantly different approach. The individual key insights and contributions are:

1. We define static program analysis problems in terms of syntactic model checking problems. In this context a bug is a violation of a syntactic model checking formula resulting in a counter-example.
2. We symbolically evaluate the feasibility of such a counter-example on a low-level program semantics using an SMT solver. If the counter-example path is infeasible we compute slices of this path that are the cause for its infeasibility and we construct an observer automaton that excludes all paths with the same cause.
3. Unlike in CEGAR we do *not* refine the whole model, but only add the observer automaton to the original model. We repeat the procedure until either all counter-examples are eliminated or a bug is found that could not be eliminated.

We evaluate our approach by applying it to a number of case studies from the NIST SAMATE program [23] and show that most of the relevant false positives can be efficiently removed using the proposed method.

Outline. In Sect. 2 we give a high-level introduction and overview of our model checking approach to static analysis as well as the ideas of the refinement loop using observers to exclude infeasible paths. We provide more details on computing infeasible sub-paths in Sect. 3 and on the construction of the observers for language refinement in Sect. 4. This is followed by large scale experiments in Sect. 5. Related work is discussed in Sect. 6. Finally, we conclude with an outlook to future work in Sect. 7.

2 Syntactic Model Checking and Language Refinement

In this section we describe our model checking approach to static program analysis and explain the key concepts of our false-positive elimination procedure. The idea of using model checking for static program analysis has first been introduced by Steffen and Schmidt [24], discussing how data flow analysis problems can be expressed in modal μ-calculus. This has later been expanded and further developed in [18,8,19].

The main idea is to abstractly represent a program (or a single function) by its control flow graph (CFG) annotated with labels representing propositions of interest. Example propositions are whether memory is allocated or freed in a particular location, whether a pointer variable is assigned $null$ or whether it is dereferenced. In this way the possibly infinite state space of a program is reduced to the finite set of locations and their propositions.

The annotated CFG consisting of the transition system and the (atomic) propositions can then be transformed into the input language of a model checker. Static analysis bug patterns can be formulated in a temporal logic and evaluated automatically by the model checker. As the annotated CFG discards most of the program semantics apart

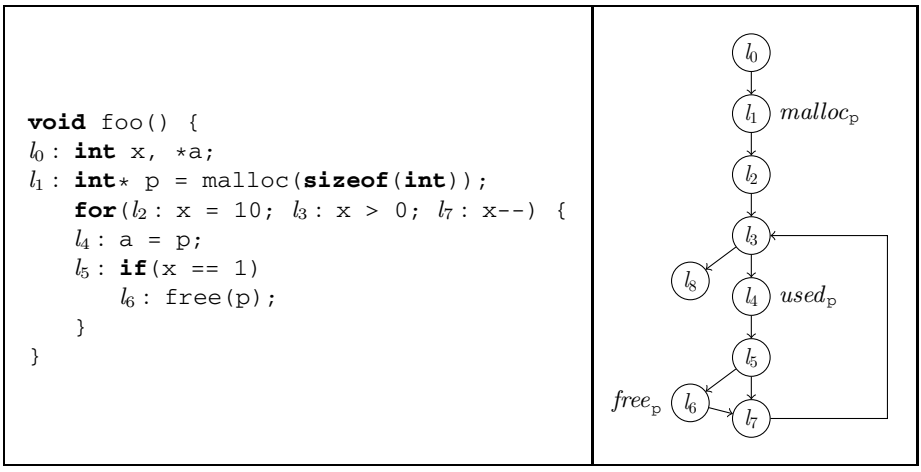


Fig. 1. Example of an annotated CFG for a function `foo`. The locations are also annotated in the listing.

from the annotations and reduces a program to its syntactical structure the approach is called *syntactic model checking* [13].

To illustrate the approach, we use a contrived function `foo` shown in Fig. 1. It works as follows: First a pointer variable `p` is initialized and memory is allocated accordingly. Then, in a loop, a second pointer variable `a` is assigned the address saved in `p`. After the tenth assignment `p` is freed and the loop is left.

To automatically check whether the memory allocated for `p` is still accessed after it is freed (a *use-after-free* in static analysis terms) we define atomic propositions for allocating memory $malloc_p$, freeing memory $free_p$ and accessing memory $used_p$, and we label the CFG accordingly. The above check can now be expressed in CTL as:

$$\mathbf{AG}(malloc_p \Rightarrow \mathbf{AG}(free_p \Rightarrow \mathbf{AG}\neg used_p))$$

This means, whenever memory is allocated, after a $free_p$ there is no occurrence of a $used_p$. Note that once a check has been expressed in CTL, the proposition can be generically pre-defined as a template of syntactic tree patterns on the abstract syntax tree of the code and determined automatically. Hence, it is possible to automatically check a wide range of programs for the same requirement. However, since our approach for false-positive elimination is based on checking path-infeasibility, we are restricted to formulas that allow linear counter-examples.

2.1 False-Positive Detection

Model checking the above property for the model depicted in Fig. 1 will find a violation and return a counter-example. The following path denoted by the sequence of locations is such a counter-example: $l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_3, l_4, l_5$.

However, if we match up the counter-example in the abstraction with the concrete program, we see that this path cannot possibly be executed, as the condition `x == 1`

Fig. 2. Parallel composition of observers with original model

cannot be true in the first loop iteration and, therefore, l_5 to l_6 cannot be taken. This means, the counter-example is spurious and should be discarded. We might get a different counter-example in the last loop iteration $\ldots, l_5, l_6, l_7, l_3, l_4, l_5$. But again, such a counter-example would be spurious, because once the condition `x == 1` holds, the loop condition prevents any further iteration.

To detect the validity of a counter-example we subject the path to a fine-grained simulation using an SMT solver. In essence, we perform a backward simulation of the path computing the *weakest precondition*. If the precondition for the initial state of the path is unsatisfiable, the path is infeasible and the counter-example spurious.

2.2 Observer Computation

Once we identified a counter-example as being spurious we know that this particular path is infeasible, but that does not mean that there are no other counter-examples for the same property. Therefore, we need to rerun the check on a refined model to see if there are other counter-examples. To get a refined model we construct a set of observer automata that have the following properties:

1. The observers can be run with the original abstract model, but they restrict the abstract model by excluding the previously computed infeasible paths.
2. The observers are based on the minimal infeasible sub-paths of a counter-example. This means, we do not need to encode each infeasible path individually, but only the set of statements that are unsatisfiable. As an example consider the assignment `x = 10` and the condition `x == 1`. Any path through these two statements, and not modifying `x` in between, will be infeasible. Hence, an observer monitoring the sub-path can be sufficient for ruling out many paths simultaneously.

Figure 2 schematically illustrates the idea of running the original model with a set of observers each representing a minimal reason for paths being infeasible. We require that in the newly composed model no observer can reach its final state, i.e., all infeasible sub-paths are excluded.

2.3 Refinement Loop

After constructing the observers based on the infeasible sub-paths, the original abstract model can be rerun to see if there are other possible counter-examples. The full path refinement loop is presented in Fig. 3. The refinement loop successively constructs new observers for new infeasible paths and extends the original model accordingly. There are two termination conditions: Firstly, we terminate whenever no bug in a program is found, i.e., there is no counter-example in the (extended) model. Secondly, we terminate

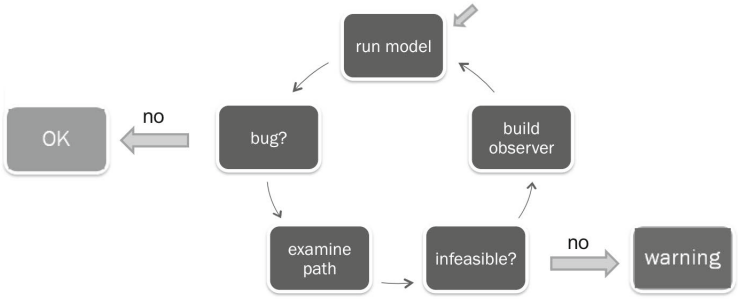


Fig. 3. Counter-example guided path refinement loop

when a path cannot be discharged as infeasible. There are two reasons for the latter: Either we found a genuine bug or our program semantics encoded in the SMT solver does not model certain aspects that are necessary to dismiss a path.

There are a few things worth noting: As we will see in the subsequent section we cover a wide variety of aspects in the C semantics including pointer aliasing. However, some constructs such as function pointers are not taken into account. In our experience, these program constructs are rarely the cause of false positives. Moreover, in the worst case we have to construct one observer for each infeasible path and there might be an exponential number of infeasible paths w.r.t. the number of conditional statements in a program. In practice, we found the number of required observers to be quite small. For most real-life cases the abstraction refinement loop terminates after two or three iterations.

2.4 A Word on SMT Solvers

In general, SMT solving tackles the satisfiability of first-order formulae modulo background theories. The approach presented in this paper is largely independent of the particular SMT solver used. However, in order to represent our semantic model of C, we require a minimum set of theories including uninterpreted functions, linear integer arithmetic and the theory of arrays, and we consider the SMT solver to support infeasible core computation.

Using additional theories (such as bit-vectors) can improve the overall precision of the presented approach for a potential penalty in runtime. We discuss the results with the given sets of theories in Sect. 5.

3 Computing Reasons for Infeasible Paths

For the path reduction refinement loop we have to identify infeasible paths. Moreover, we are interested in a small sequence of statements that explains why a path is infeasible. Such an explanation will allow us to exclude all paths with that infeasible sequence of statements. For instance, in the path through $l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_3, l_4$ of `foo` in Fig. 1 not all statements are contributing to it being infeasible, but only l_2 : `x = 10`

and (l_5, l_6) : `x == 1`. We call the sequence of edges $(l_2, l_3), (l_3, l_4), (l_4, l_5), (l_5, l_6)$ an *infeasible sub-path.*

Next, we explain how to detect infeasible paths in a C program by means of satisfiability checking of weakest preconditions using an SMT solver. Moreover, we provide a strategy to efficiently compute an infeasible sub-path from an infeasible path, which enables the construction of an efficient observer.

3.1 Detecting Infeasible Paths

For checking the feasibility of a path we first collect the sequence of statements in that path along the CFG. Moreover, we encode branching decisions along that path as assertions in the sequence of statements, resulting in a straight-line program. Next, we compute the weakest precondition for this straight-line program. The SMT solver might return that the weakest precondition is unsatisfiable, i.e., that the path cannot be executed and, therefore, is infeasible. In the following we provide the basic ideas for modeling program statements, their semantics and the representation in an SMT solver. The details of the underlying semantics and the memory model can be found in [21].

Path programs. Computing the straight-line program corresponding to a path through the CFG amounts to collecting the sequence of assignments and function calls taken on the path. Additionally, `assert` statements record what must be true at a point of execution to follow the path through control-flow statements, like taking the then-branch of an `if`. For example, the path $(l_2, l_3), (l_3, l_4), (l_4, l_5), (l_5, l_6), (l_6, l_7)$ of the CFG of `foo` in Fig. 1 is represented by the path program

```
x = 10; assert(x > 0); a = p; assert(x == 1); free(p);
```

More formally, a *path program* is a sequence of statements s containing expressions e. A statement can be e_1 `=` e_2`;` for assignments, `assert(`e`);` for checking a (boolean) condition and f`(`e_1`, ..., `e_n`);` respectively e_0 `=` f`(`e_1`, ..., `e_n`);` for function calls (optionally assigning the return value). In our approach, an expression may be almost any valid C-expression [20] including pointers and using `structs`. Currently, however, we do not support function pointers, and string literals are treated as fresh pointer variables. We make the simplifying assumption that identifiers such as program variables and field names of `structs` are globally unique.

Weakest precondition. The set of states from which a path program can be executed without violating any assertion is given by the weakest precondition of the path program w.r.t. the trivial postcondition $true$. Generally, the *weakest precondition* $wp(p, \psi)$ of a path program p w.r.t. a condition ψ on states is given by a condition φ which is satisfied by exactly those states from which an execution of p terminates in a state satisfying ψ. In particular, $wp($`assert(`e`);`$, \psi)$ is equivalent to $e \wedge \psi$, indeed asserting that e must already hold.

The computed formula $wp(p, true)$ characterizing successful executability of p will be handed to an SMT solver for checking unsatisfiability. However, we will not always be able to represent executability faithfully in terms of a full C-semantics, but may

have to use safe approximations. These approximations have to ensure (under certain assumptions) that the unsatisfiability of $wp(p, true)$ implies that p is not executable.

In particular, our definition of wp is based on a simple memory model that allows a good precision even in the presence of variable aliasing and basic pointer arithmetic. The memory model is similar to the one described by Burstall [5]. The main idea is to have a separate store, represented by a separate variable, for each primitive data type. We use a simple type system consisting of primitive types like integers and pointer types as well as `struct`-like composite types. Each store is again segmented into partitions, one for each field of a `struct` and a distinguished partition for data that is not part of a `struct`. The rationale behind this kind of model is that by introducing logical structure such as distinct memories we achieve the property that certain aliasing is not possible, such as aliasing between variables of different types or between different fields of a `struct`. Properties that we do not get by construction are enforced by axioms. An example for such an axiom is that local variables do not alias.

The memory model can be easily encoded in a language for SMT solvers. In our experiments we used the theory of arrays [25]. The theory provides two operations: $access(m, a)$ (sometimes called $read$ or $select$) to access the value of an array m at location a and $update(m, a, v)$ (sometimes called $write$ or $store$) to get a version of m that is updated at location a with value v. Using the theory of arrays, the memory model can be represented as an array with tuples (containing a partition name and a position) as indices.

To illustrate the use of the memory model in the wp-semantics we consider a simple assignment `x = v`, where the value of local variable `x` of type τ is assigned the value of a local variable `v`, also of type τ. The wp-semantics in this case is

$$wp(\texttt{x = v}, \varphi) = \varphi\{M_\tau \mapsto update(M_\tau, loc(\texttt{x}), v)\} \wedge v = access(M_\tau, loc(\texttt{v}))$$

where M_τ is the memory variable for τ and loc is a function mapping a variable name to a location (i.e. partition and position).

A limitation with regard to the definition of wp are function calls. We currently do not consider the true effect of called functions, but approximate the effect by assuming that only those locations in the memory are touched that are explicitly passed as a pointer. An exception in this regard is the `malloc` function. As it is central to handle pointers sufficiently precise we use axioms to specify its semantics. An example for such an axiom is that `malloc` always returns fresh memory locations, which are not aliased with any other. On the other hand, no special axioms are needed for `free`, ignoring whether its argument points to allocated memory. More details on the wp-semantics can be found in [21].

Infeasible Paths. Based on the weakest precondition semantics, infeasibility follows naturally as: A path through the CFG is called *infeasible* if $wp(p, true)$ for its corresponding path program p is unsatisfiable.

For example, the path $(l_2, l_3), \ldots, (l_6, l_7)$ from above is infeasible since

$$wp(\texttt{x = 10; assert(x > 0); a = p; assert(x == 1); free(p);}, true)$$

is unsatisfiable due to incompatibility of `x = 10` and `x == 1`. Next, we explain how to identify shorter sub-paths capturing the relevant causes for infeasible paths.

3.2 Computing Infeasible Sub-paths

The general idea of this work is to create observers to exclude infeasible paths in static program analysis. We would like, however, to avoid to generate one observer for each path, but instead to identify smaller sub-paths that capture unsatisfiable inconsistencies. Excluding these sub-paths might exclude a wider set of paths passing through those fragments and, as a result, one observer will be able to exclude many infeasible paths at once. For instance, the path $(l_2, l_3), \ldots, (l_6, l_7)$ above shows that with respect to infeasibility it is irrelevant how an execution reaches l_2 : `x = 10` and how it continues after (l_5, l_6) : `assert(x == 1);`: Due to the incompatibility of `x = 10` and `x == 1` on that path, any path containing $(l_2, l_3), (l_3, l_4), (l_4, l_5), (l_5, l_6)$ as a sub-path is infeasible.

Generally, let us say that path π' is a *sub-path* of path π and π *includes* π' if π is of the form $\pi_0\ \pi'\ \pi_1$ for some (possibly empty) paths π_0 and π_1. This leads to:

Proposition 1. *Every path including an infeasible sub-path is infeasible.*

Thus, if we find an infeasible sub-path in a path from a counter-example, we can exclude all paths that include this sub-path. We can compute the infeasible sub-paths by computing the unsatisfiable sub-formulae of the weakest precondition. In order to match sub-formulae and sub-paths we split the weakest precondition into named conjuncts each of which corresponds to a statement in the considered path (see [21] for details).

SMT solvers usually can be instructed to deliver an unsatisfiable sub-formula. It is advantageous to identify small unsatisfiable sub-formulae leading to short infeasible sub-paths, thus allowing to exclude potentially more paths. However, finding all minimal unsatisfiable sub-formulae requires exponentially many calls to the SMT solver in the worst case (for algorithms see, e.g., [9] and [22]). We therefore heuristically enumerate unsatisfiable sub-formulae using the solver and employ an exponential algorithm only to minimize these [21].

4 Observer Construction and Refinement

In this section we formally define how to construct observers based on sub-paths. Moreover, we show how to compose the observers with the original model in a refinement loop for eliminating false positives.

In short, for the observer construction we view a CFG as a finite automaton that accepts paths as sequences of edges through the CFG as words. From an infeasible sub-path we construct an "observing" finite non-deterministic automaton. The language of this observing automaton is the set of paths which include the infeasible sub-path. We consider the synchronous product of the CFG automaton and the complemented observing automaton, where synchronization is on the shared alphabet, i.e., the edges. This product automaton accepts exactly those paths as sequences of edges that do not show an infeasible sub-path.

4.1 Representing Programs as Automata

We rely on the conventional notion of a finite (non-deterministic) automaton $M = (A, S, T, I, F)$ consisting of an alphabet A, a finite set of states S, a transition relation

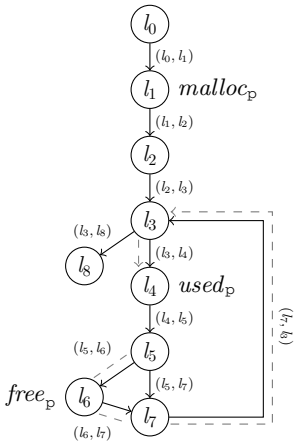


Fig. 4. CFG automaton for the function `foo`. The dashed edges represent an infeasible sub-path.

$T \subseteq S \times A \times S$, a set of initial states $I \subseteq S$, and a set of final states $F \subseteq S$. The words accepted by M are denoted by $\mathcal{L}(M)$. We write $M \times N$ for the synchronous product of the finite automata M and N over the same alphabet; then $\mathcal{L}(M \times N) = \mathcal{L}(M) \cap \mathcal{L}(N)$ holds. The finite automaton yielding language complement is denoted by M^{c}, i.e., $\mathcal{L}(M^{\mathrm{c}}) = A^* \setminus \mathcal{L}(M)$ where A is the alphabet of M.

A CFG can naturally be regarded as such a finite automaton with the states being the locations. For the alphabet we choose pairs of locations(l, l'), i.e., making the edges of the CFG "observable". The transition relation of the automaton just follows from the CFG. All the states are both initial and final to capture arbitrary sub-paths in the CFG.

Definition 1 (CFG Automaton). *For a CFG with locations L and edges $E \subseteq L \times L$, its corresponding* CFG automaton *is the finite automaton given by (E, L, T, L, L), where the alphabet is the set of edges E, the states are the locations L, the transition relation is $T = \{(l, (l, l'), l') \mid (l, l') \in E\}$, and all states are both initial and final.*

The words accepted by a CFG automaton correspond exactly to the paths as sequences of control-flow edges through the CFG. Therefore, we will also call these accepted words "paths". The CFG automaton for the function `foo` is shown in Fig. 4.

A CFG automaton can also be directly used for model-checking, as the annotations of the CFG such as $malloc_{\mathrm{p}}$ can be interpreted as predicates over its states. For `foo` we would define $malloc_{\mathrm{p}} \equiv l_1$ or $used_{\mathrm{x}} \equiv l_3 \vee l_5$.

4.2 Computing Observers from Counter-Examples

If the model checking procedure yields a counter-example as a path through a CFG automaton, which is infeasible, we want to exclude this path in further model checking runs. In fact, the notion of infeasible sub-paths allows us to exclude all paths that include some infeasible sub-path due to Prop. 1. Consider, for example, the CFG automaton in Fig. 4. The dashed edges represent an infeasible sub-path $\pi = (l_5, l_6), (l_6, l_7), (l_7, l_3),$

(l_3, l_4) of an infeasible counter-example reported by the model checker. We can not only exclude π but also a path that represents a two-fold loop iteration and then continues as before. On the other hand, we cannot exclude a path that has (l_5, l_7) instead of $(l_5, l_6), (l_6, l_7)$.

For a sub-path π accepted by the CFG automaton, we construct an automaton that accepts exactly those paths π' for which π is a sub-path. We define:

Definition 2 (Observer). *Let P be a CFG automaton with alphabet E and let $\pi = e_1 \dots e_k$ be a path accepted by P. The* CFG observer automaton *$Obs(E, \pi)$ is the automaton (E, S_{Obs}, T, S_0, F), where*

- *S_{Obs} is the set of states $\{\mathsf{s}_1, \dots, \mathsf{s}_{k-1}\} \cup \{\mathsf{Init}, \mathsf{Infeasible}\}$.*
- *$T \subseteq S_{Obs} \times E \times S_{Obs}$ is the transition relation. A triple (s, e, s') is in the relation if and only if one of the following holds:*
  1. *$s = \mathsf{Init}$ and $s' = \mathsf{Init}$ and $e \neq e_1$*
  2. *$s = \mathsf{s}_i$ and $s' = \mathsf{s}_{i+1}$ and $e = e_{i+1}$ and $1 \leq i \leq k-2$*
  3. *$s = \mathsf{s}_{k-1}$ and $s' = \mathsf{Infeasible}$ and $e = e_k$*
  4. *$s \neq \mathsf{Infeasible}$ and $s' = \mathsf{s}_1$ and $e = e_1$*
  5. *$s = \mathsf{s}_i$ and $s' = \mathsf{Init}$ and $e \in E \setminus \{e_1, e_{i+1}\}$ and $1 \leq i \leq k-1$*
  6. *$s = \mathsf{Infeasible}$ and $s' = \mathsf{Infeasible}$*
- *$S_0 = \{\mathsf{Init}\}$ is the set of initial states.*
- *$F = \{\mathsf{Infeasible}\}$ is the set of final states.*

The rationale for the particular choice of the observer's components is as follows: The states mirror how much of π has already been observed on a run without interruption. When the observer is in state Init, nothing has been observed at all or a part of π has been observed, but then the sequence was interrupted. If the observer is in state Infeasible the whole path π has already been observed, which means no matter how the program model continues, the current run already represents an infeasible path. If the automaton is in state s_i, we know π has been observed until and including e_i. The transition relation reacts to an edge on the run:

1. As long as the initial edge e_1 of π has not been observed, the observer needs to stay in Init.
2. If the observer has already observed the first i edges of π and now observes the next edge e_{i+1} it proceeds one step further, as long as e_{i+1} is not the last edge of π.
3. If the situation is as in (2) but e_{i+1} is the last edge of π, the observer transitions to Infeasible.
4. It may happen that the observer already is in state s_j when another sequence of π starts. Intuitively, π is interrupted by itself. Therefore the observer may transition to s_1 as soon as it observes e_1, even if it is currently in some s_j.
5. If the sequence is interrupted in a different way, the observer returns to Init.
6. As soon as the observer is in state Infeasible, it remains there forever.

Example 1. We illustrate the observer construction with our running example. Regarding the CFG automaton of the function `foo`, a path containing the sub-path $\pi = (l_5, l_6), (l_6, l_7), (l_7, l_3), (l_3, l_4)$ is infeasible. The constructed observer automaton is depicted in

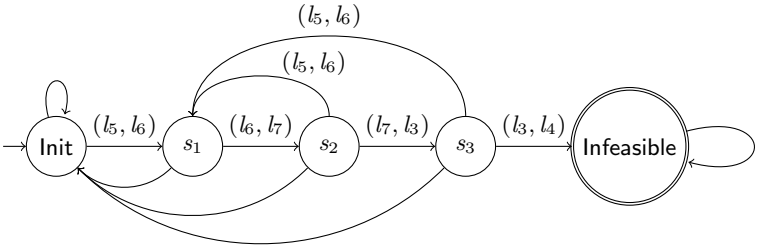


Fig. 5. Observer automaton for infeasible path of Example 1. Unlabeled edges mean "any other".

Fig. 5. As soon as it observes the sequence π it enters the state Infeasible and remains there forever. If the sequence is interrupted, it either returns to Init or, if the interruption equals (l_5, l_6) as the first edge of π, it returns to s_1. Hence, as long as the observer is not in state Infeasible the sequence π has not been observed completely. As Infeasible is the only accepting state, the observer only accepts paths that contain π, i.e., infeasible paths.

Let π denote a path accepted by a CFG automaton P with the alphabet E of control-flow edges and let $\mathcal{P}(\pi)$ be the set of paths in E^* including π. By construction, we have $\mathcal{P}(\pi) = \mathcal{L}(Obs(E, \pi))$, that is, the words accepted by $Obs(E, \pi)$ are exactly the paths including π. Furthermore,

$$\mathcal{L}(P \times Obs(E, \pi)^{\mathrm{c}}) = \mathcal{L}(P) \cap \mathcal{L}(Obs(E, \pi)^{\mathrm{c}}) = \\ \mathcal{L}(P) \cap (E^* \setminus \mathcal{L}(Obs(E, \pi))) = \mathcal{L}(P) \cap (E^* \setminus \mathcal{P}(\pi)) \,.$$

Thus by applying Prop. 1, that all paths including an infeasible sub-path are infeasible, we get

Proposition 2. *Let P be a CFG automaton P with alphabet E and let π be an infeasible path of P. Then the CFG automaton resulting from the synchronous product of P and $Obs(E, w)^{\mathrm{c}}$ excludes the infeasible paths that include π.*

4.3 Implementing Observers

The observer is in general non-deterministic. Computing the complement of a non-deterministic automaton would involve first creating its deterministic equivalent, which can have exponential size compared with the non-deterministic automaton. We avoid directly constructing the complement of the observer and instead implement the complementation by adding a fairness constraint in the model checker [14]. The fairness constraint in our case forbids that the observer enters state Infeasible. Although fair CTL model checking is more complex than regular CTL model checking, it works well in our experiments, as the next section shows.

5 Experiments

In this section we report on the implementation of the aforementioned false-positive elimination techniques as well as on analysis results from representative, large code

bases. All the experimental data has been obtained from projects and benchmarks provided by NIST and the Department of Homeland Security for the 2010 and 2011 Static Analysis Tool Exposition (SATE) [23]. The experiments show that the proposed solution provides a significant decrease in false positives while only moderately increasing the overall runtime.

5.1 Implementation

We implemented a prototype of the SMT-based path reduction approach in our static analysis tool Goanna[1]. Goanna is a state-of-the-art static analysis tool for bug detection and security vulnerability analysis of industrial C/C++ programs. Currently, Goanna supports around 150 classes of different checks ranging from memory leak detection and *null*-pointer dereferences to the correct usage of copy control in C++ as well as buffer overruns.

The Goanna tool itself as well as the new false -positive elimination procedure is implemented in the functional programming language OCaml. For the infeasible path detection in our experiments we used the Z3 SMT solver [10], as it provides an OCaml interface which allowed quick prototyping using Goanna.

5.2 Experimental Evaluation

As representative test beds for our experiments we chose the two main open source projects from the NIST SATE 2010 and 2011 exposition: Wireshark 1.2.9 and Dovecot 2.0 beta6. Wireshark is a network protocol analyzer consisting of around 1.4MLoc of pure C/C++ code that expand to roughly 16MLoc after pre-processing (macro expansions, header file inclusion etc.). Dovecot is a secure IMAP and POP3 server that consists of around 170KLoc of pure C/C++ code expanding to 1.25MLoc after preprocessing. We experimented with other in-house industrial code of different sizes as well and obtained very similar results as for the two mentioned projects.

The evaluation was performed on a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at 3.4GHz, 2MB L2 cache and 1.5GB DDR-2 400MHz ECC memory.

False-Positive Removal Rates. As mentioned earlier, Goanna performs a source code analysis for around 150 classes of checks. However, not all checks are path-sensitive, i.e., some checks only require tree-pattern matching, and of those checks that are path-sensitive not all are amenable to false path elimination. The reasons are as follows: Certain path-sensitive checks such as detecting unreachable code already state that there is no path satisfying a certain requirement. Hence, removing infeasible paths will not change the results. A similar example is having no path where a `free()` occurs after a `malloc()` and alike. The results below only include checks where false path elimination can alter the analysis results.

The false-positive elimination results for Wireshark and Dovecot are summarized in Table 1. For Wireshark, our original Goanna implementation detected 98 relevant

[1] http://www.nicta.com.au/goanna

Table 1. False Positive Detection Rate for Wireshark and Dovecot

	Wireshark 1.2.9	Dovecot 2.0 beta6
lines of code	1,368,222	167,943
after pre-processing	16,497,375	1,251,327
number of functions	52,632	5,256
issued warnings	98	75
false positives removed	48	38
% removed warnings	49.0%	50.6%
correctly identified false positives	48 (100%)	38 (100%)

Table 2. Runtime Performance for False-Positive Elimination

	Wireshark 1.2.9	Dovecot 2.0 beta6
total running time (no timeout)	8815s	1025s
time spent in refinement loop	1332s (15%)	302s (29.5%)
% of time in SMT	10.5%	12.2%
% of time in model checking	87.5%	86.3%
number of Goanna timeouts	12	1
number of SMT loops exceeding (20)	11	3
number of SMT solver timeouts	0	5

path-sensitive issues. Running Goanna with the new SMT-based false path elimination approach removed 48 issues. This means, around 49% of the produced warnings were eliminated fully automatically. We manually investigated all of the removed warnings and were able to confirm that these were indeed all false positives.

The results for Dovecot are very similar to the Wireshark results. The original implementation raised 75 warnings and we were able to automatically identify 38 of those warnings as false positives. This means, the number of warnings was reduced by 50.6%. Again, all the automatically removed warnings were confirmed false positives.

For both projects, we investigated the remaining issues manually in detail. There were several remaining false positives for various reasons: Due to the incompleteness of the procedure, e.g., missing further knowledge about functions calls, a path could not be identified as infeasible. Another reason is that we imposed a loop limit of 20 refinement iterations. Sometimes this limit was reached before a warning could be refuted. This happened 11 times in Wireshark, but only 3 times in Dovecot. As a side note, as discussed in [15], there are in general various reasons for false positives and often additional context information known to the developer is the key for refuting false positives. Moreover, it is worth noting that for path-insensitive checks (e.g., pattern matching) the number of false positives tends to be much lower or even zero.

Run-time Performance. The runtime results for the experiments are shown in Table 2. For the experiment we introduced timeouts both in Goanna as a whole as well as the SMT solver. For Goanna including the SMT path reduction loop an upper limit of 120s per file was set and in the SMT solver of 2s per solving. Moreover, we limited the maximum depth of SMT loops by 20. The timeouts, however, were only triggered very sporadically: Goanna timeouts occurred 12 times in Wireshark and once in Dovecot,

which in both projects accounts for roughly 0.02% of all functions. Loop limits were reached similarly often and SMT timeouts occurred never in Wireshark and 5 times in Dovecot. In the remainder the analysis results are based on all non-timeout runs.

As shown in Table 2 the overall runtime for Wireshark was around two and a half hours, for Dovecot around 17min. In Wireshark for checks that can be improved through false path elimination around 15% of the runtime was spent in the SMT refinement loop. For the same objective the overhead in Dovecot was slightly higher with around 30%.

Interestingly, the vast majority of the overhead time is spent in the repeated model checking procedure rather than the SMT solving. Although the additional observers increase the state space in theory, the reachable state space will always be smaller than in the original model, since the observers constrain the set of reachable states. We have since then identified unnecessary overheads in our model checking procedure that should reduce the overall runtime in the future. However, given the value of a greatly reduced number of false positives, which can otherwise cost many engineering hours to identify, we believe that a run-time overhead of 15%–30% is already acceptable in practice; especially, if it equates to around 22min in over one million lines of C/C++ code.

6 Related Work

Counter-example based path refinement with observers for static program analysis has been introduced by Fehnker et al. [14]. This work was based on using interval abstract interpretation to refute infeasible paths. While fast, it was limited to simple root causes for infeasible paths and much less precise than the SMT approach in this work. On the other hand, the application of predicate abstraction in conjunction with on-demand refinement has long been present in the CEGAR [6] approach and is used in many software model checkers such as SLAM [17] and BLAST [4,3]. This approach refines the whole model iteratively instead of eliminating sets of paths and using observers to learn from it. To an extent, a comparison of both approaches is still missing given their origin from different domains, namely static analysis and software mode checking.

The detection of infeasible paths and its use for program analysis has been explored by other authors, as well. Balakrishnan et al. [2] use this technique in the context of abstract interpretation. Delahaye et al. [11] present a technique how to generalize infeasible paths, but they have not investigated its use in static analysis. Yang et al. [26] propose the use of SMT solvers to remove infeasible paths by Dynamic Path Reduction. However, the work only addresses programs without pointers employing standard weakest precondition and it is not aimed at false-positive elimination. Harris et al. [16] describe a way to do program analysis by enumerating path programs. In contrast to our work they are not in a model-checking setting and their approach is not driven by counter-examples.

Finally, there are many examples of using SMT solvers in the realm of software model checking, e.g., as reasoning engine for bounded model checking [1,7].

7 Conclusions and Future Work

We have introduced a novel approach to reducing false positives in static program analysis. By treating static analysis as a syntactical model checking problem, we make static

analysis amenable to an automata-based language refinement. Moreover, unlike traditional CEGAR approaches we create observer automata that exclude infeasible subpaths. The observers are computed based on a weakest precondition semantics using an SMT solver. We have shown that the approach works very well in practice and detects many relevant false positives.

Future work will further explore the limits of false-positive removal. We plan to investigate if more expensive SMT theories will lead to more false-positive removals or if, in fact, there are hardly any cases where this is necessary. Also, we will focus on further comparison with existing software model checking approaches and investigate if we can "out-source" some false-positive removal directly to a software model checker without much runtime penalty.

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. Int. J. Softw. Tools Techn. Transf. 11(1), 69–83 (2009)
2. Balakrishnan, G., Sankaranarayanan, S., Ivančić, F., Wei, O., Gupta, A.: SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 238–254. Springer, Heidelberg (2008)
3. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic Predicate Abstraction of C Programs. In: Proc. 2001 ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI 2001), pp. 203–213. ACM (2001)
4. Ball, T., Rajamani, S.K.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
5. Burstall, R.: Some Techniques for Proving Correctness of Programs which Alter Data Structures. Mach. Intell. 7, 23–50 (1972)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. In: Proc. 24th IEEE/ACM Int. Conf. Automated Software Engineering (ASE 2009), pp. 137–148. IEEE (2009)
8. Dams, D.R., Namjoshi, K.S.: Orion: High-Precision Methods for Static Error Analysis of C and C++ Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 138–160. Springer, Heidelberg (2006)
9. de la Banda, M., Stuckey, P., Wazny, J.: Finding All Minimal Unsatisfiable Subsets. In: 5th Int. ACM SIGPLAN Conf. Principles and Practice of Declarative Programming (PPDP 2003), pp. 32–43. ACM (2003)
10. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
11. Delahaye, M., Botella, B., Gotlieb, A.: Explanation-Based Generalization of Infeasible Path. In: Proc. 3rd Int. Conf. Software Testing, Verification and Validation (ICST 2010), pp. 215–224. IEEE (2010)
12. D'Silva, V., Kroening, D., Weissenbacher, G.: A Survey of Automated Techniques for Formal Software Verification. IEEE Trans. CAD Integ. Circ. Syst. 27(7), 1165–1178 (2008)
13. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model Checking Software at Compile Time. In: Proc. 1st Joint IEEE/IFIP Symp. Theoretical Aspects of Software Engineering (TASE 2007), pp. 45–56. IEEE (2007)

14. Fehnker, A., Huuck, R., Seefried, S.: Counterexample Guided Path Reduction for Static Program Analysis. In: Dams, D., Hannemann, U., Steffen, M. (eds.) Concurrency, Compositionality, and Correctness. LNCS, vol. 5930, pp. 322–341. Springer, Heidelberg (2010)
15. Fehnker, A., Huuck, R., Seefried, S., Tapp, M.: Fade to Grey: Tuning Static Program Analysis. In: Proc. 3rd Int. Wsh. Harnessing Theories for Tool Support in Software (TTSS 2009), pp. 38–51. UNU-IIST (2009)
16. Harris, W.R., Sankaranarayanan, S., Ivančić, F., Gupta, A.: Program Analysis via Satisfiability Modulo Path Programs. In: Proc. 37th ACM SIGPLAN-SICACT Symp. Principles of Programming Languages (POPL 2010), pp. 71–82. ACM (2010)
17. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
18. Holzmann, G.J.: Static Source Code Checking for User-Defined Properties. In: Proc. 6th World Conf. Integrated Design and Process Technology (IDPT 2002). SDPS (2002)
19. Huuck, R., Fehnker, A., Seefried, S., Brauer, J.: Goanna: Syntactic Software Model Checking. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 216–221. Springer, Heidelberg (2008)
20. ISO/IEC. ISO/IEC 9899:2011 Information Technology – Programming Languages – C. ISO, Genève (2011)
21. Junker, M.: Using SMT Solvers for False Positive Elimination in Static Program Analysis. Master's thesis, Universität Augsburg (2010), http://www4.in.tum.de/~junkerm/publications/thesis.pdf
22. Liffiton, M.H., Sakallah, K.A.: On Finding All Minimally Unsatisfiable Subformulas. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 173–186. Springer, Heidelberg (2005)
23. Okun, V., Delaitre, A., Black, P.E. (eds.): Report on the Third Static Analysis Tool Exposition (SATE 2010). SP-500-283, U.S. Nat. Inst. Stand. Techn. (2011)
24. Schmidt, D.A., Steffen, B.: Program Analysis *as* Model Checking of Abstract Interpretations. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 351–380. Springer, Heidelberg (1998)
25. Stump, A., Barrett, C., Dill, D., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: Proc. 16th Ann. IEEE Symp. Logic in Computer Science (LICS 2001), pp. 29–37. IEEE (2001)
26. Yang, Z., Al-Rawi, B., Sakallah, K., Huang, X., Smolka, S., Grosu, R.: Dynamic Path Reduction for Software Model Checking. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 322–336. Springer, Heidelberg (2009)

Predicate Analysis with Block-Abstraction Memoization*

Daniel Wonisch and Heike Wehrheim**

University of Paderborn, Germany
wehrheim@upb.de

Abstract. Predicate abstraction is an established technique for reducing the size of the state space during verification. In this paper, we extend predication abstraction with *block-abstraction memoization* (BAM), which exploits the fact that blocks are often executed several times in a program. The verification can thus benefit from *caching* the values of previous block analyses and reusing them upon next entry into a block. In addition to function bodies, BAM also performs well for *nested loops*. To further increase effectiveness, block memoization has been integrated with *lazy abstraction* adopting a lazy strategy for cache refinement. Together, this achieves significant performance increases: our tool (an implementation within the configurable program analysis framework CPAchecker) has won the Competition on Software Verification 2012 in the category "Overall".

1 Introduction

In recent years, software model checking has become an effective technique for software verification. This success is due to the tight integration of elaborate program analysis and model checking techniques, and the recent advances in SMT solving. Software model checking tools like SLAM [1], BLAST [9], ARMC [23], SATABS [13] or F-Soft [20] are used to analyze industrial size programs. Still, scalability is one of the major issues and the driving force for research in this area.

A large number of today's software model checkers employ abstraction concepts in their analysis, more precisely *predicate abstraction* [16]. Predicate abstraction builds an abstract representation of a program in which the basic control flow is maintained and the concrete states are replaced by predicates on states. The level of abstraction needed for a particular analysis is incrementally determined, starting with a coarse abstraction which is then refined based on computed (possibly spurious) counter examples. Elaborate techniques like lazy abstraction [19] and Craig interpolation [18] help to keep the abstraction small, the former by adding new predicates to relevant program positions only and the latter by computing a parsimonious set of new predicates.

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

** Corresponding author.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 332–347, 2012.

Predicate abstraction techniques often represent abstractions in the form of an *abstract reachability tree* (ART). An ART can be seen as an unfolding of the control flow graph in which nodes describe program locations together with abstract states (evaluations of predicates) and edges correspond to statements (or possibly blocks) of the program. Statements which can be reached several times during program execution will appear several times in the ART. This approach has two drawbacks. On the one hand, the effect of these statements on the abstract state needs to be recomputed every time they are seen during the unfolding, even though they might modify the abstract state in the same way. On the other hand, the ART might get rather huge due to these repeated occurrences of statements. This is in particular the case when we have statements belonging to (a) functions being called several times, or (b) nested loops.

In this paper, we propose *block-abstraction memoization* (BAM) which overcomes both of these drawbacks. Block-abstraction memoization considers entire blocks of statements, for instance bodies of functions or loops. During construction of the ART, BAM computes a separate ART for each block. These "block ARTs" only refer to *locally* relevant variables. A block ART acts as a cached value of a block's behavior. The main ART just gets a "block" edge for the entire block. Whenever the same block is reached again, the already explored block behavior stored in the block ART can be reused - given that the current abstract state projected onto the block's locally relevant variables is the same.

To further increase the effectiveness, BAM is integrated with lazy abstraction. Lazy abstraction uses different precisions (predicate sets) at different abstract states. Combined with block-abstraction memoization the challenge is to determine when, where, and how to refine a cached block abstraction. For this, we have decided – after experimentation with several approaches – to employ a *lazy strategy*: already cached block abstractions are only refined where needed, i.e., when touched by a counter example and only on the path of the counter example in the ART. This necessitates rebuilding of blocks ARTs at some places, however, the performance decrease by this reconstruction is in almost all case studies more than outruled by the performance gain of faster abstraction refinement.

The idea of summarizing the input-output behaviour of *procedures* with respect to certain relevant information is an established technique in static analysis and has also been studied for model checking (e.g. Bebop [4], Yogi [15]). With respect to lazy abstraction, only [15] integrates summaries with a lazy technique, however, without refining summaries.

We implemented our approach within the configurable program analysis framework CPAchecker [10], which carries out a predicate analysis using lazy abstraction. We evaluated our approach on the benchmark C programs of the Competition on Software Verification 2012 [8], which we won in the category "Overall" [26]. This convincingly demonstrates the usefulness of BAM, not just for specific programs with lots of functions, but for a broad range of C programs from a standard benchmark set.

2 Preliminaries

In the following we briefly introduce the preliminaries needed to understand the presented technique. As we implemented our approach within the program analysis framework CPAchecker, we use the formalism of configurable program analysis (CPA) [6] for the presentation of our technique, however largely eliding those parts referring to CPAchecker's configuration and precision adjustment features. Instead we focus on the part necessary for explaining our own approach.

Concrete Semantics of Programs. For the sake of presentation we consider programs to be written in a simple imperative programming language with assignments and assumes operation as only possible statements, and variables that range over integers only. The left of Figure 1 shows our running example of a program consisting of three nested loops. In the program each loop increments a respective counter variable twice. After the execution of all loops it is asserted that the counting variables of the loops are indeed 2. We assume that `//do something` inside the innermost loop does not refer to i and j. This is of course unrealistic; the example is merely chosen for didactic purposes. Note that the experimental results show that our technique also works quite well for standard C benchmark programs.

Formally, a *program* $P = (A, l_0, l_E)$ is represented as *control-flow automaton* (CFA) A, a start location l_0, and an error location l_E. A CFA $A = (L, G)$ consists of a set of (program) locations L and a set of edges $G \subseteq L \times Ops \times L$ that describe possible transitions from one program location to another by executing a certain operation $op \in Ops$. The right of Figure 1 shows the control-flow automaton of program `NESTED` (dotted rectangles denote blocks; see next section).

```
l0:  int i=2, j=2, k=2;
l1:  i = 0;
l2:  while(i < 2) {
l3:    j = 0;
l4:    while(j < 2) {
l5:      k = 0;
l6:      while(k < 2) {
           //do something
l7:        k++;}
l8:      j++;}
l9:    i++;}
l10: assert(i==2 && j==2 && k==2)
```

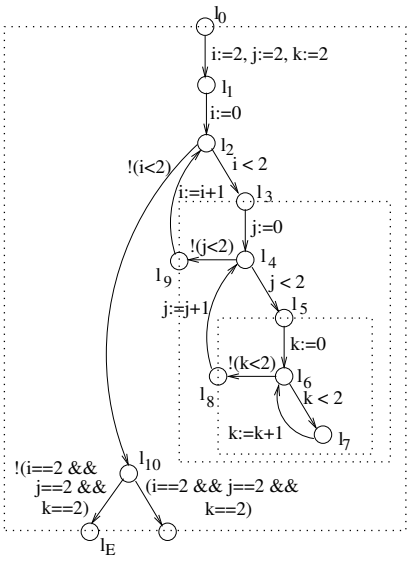


Fig. 1. Program `NESTED` and its control-flow automaton with 3 blocks

A *concrete data state* $c : X \to \mathbb{Z}$ of a program P is a mapping from the set of variables X of the program to integer values. The set of all concrete data states in a program P is denoted by $\mathscr{C}$. A set of concrete data states, also called *region*, can be described by a first-order predicate logic formula φ. We write $[\![\varphi]\!] := \{c \in \mathscr{C} \mid c \models \varphi\}$ for the set of concrete data states represented by some formula φ. Furthermore, we write $\gamma(c)$ for the representation of a concrete data state as formula (i.e. $[\![\gamma(c)]\!] = \{c\}$).

A tuple (l, c) of a location and a concrete data state of a program is called *concrete state*. The *concrete semantics* of an operation $op \in Ops$ is defined in terms of the *strongest postcondition operator* $SP_{op}(\cdot)$. Intuitively, the strongest postcondition operator $SP_{op}(\varphi)$ of a formula φ wrt. to an operation op is the strongest formula ψ which represents all states which can be reached by op from a state satisfying φ. Formally, we have $SP_{x:=expr}(\varphi) = \exists \widehat{x} : \varphi_{[x \mapsto \widehat{x}]} \wedge (x = expr_{[x \mapsto \widehat{x}]})$ for an assignment operation $x := expr$ and $SP_{assume(p)}(\varphi) = \varphi \wedge p$ for an assume operation $assume(p)$ ($assume(\cdot)$ omitted in figures).

We write $(l, c) \xrightarrow{g} (l', c')$ for concrete states (l, c), (l', c') and edge $g := (l, op, l')$, if $c' \in [\![SP_{op}(\gamma(c))]\!]$. We write $(l, c) \to (l', c')$ if there is an edge $g = (l, op, l')$ such that $(l, c) \xrightarrow{g} (l', c')$. A concrete state (l_n, c_n) is *reachable* from a location l_0 and a region r, denoted by $(l_n, c_n) \in Reach(\{l_0\} \times r)$, if there is a concrete data state $c_0 \in r$ such that $(l_0, c_0) \to (l_1, c_1) \to \ldots \to (l_n, c_n)$ for some intermediate concrete states $(l_1, c_1), \ldots, (l_{n-1}, c_{n-1})$. Sometimes we like to explicitly refer to a CFA A when talking about the set of reachable states, which we denote as $Reach_A(\cdot)$. Furthermore, we say a location l' is reachable in a program $P = (A, l_0, l_E)$ if there is a concrete state $(l', c') \in Reach(\{l_0\} \times \mathscr{C})$. Finally, a program $P = (A, l_0, l_E)$ is *safe* if l_E is not reachable in P.

Predicate Abstraction. Let $\mathscr{P}$ be a set of quantifier-free predicates over program variables. A *predicate formula* φ is a boolean combination of predicates from $\mathscr{P}$. A *precision for formulas* $PS \subseteq \mathscr{P}$ is a finite subset of $\mathscr{P}$ with which a predicate formula can be constructed. A *precision for programs* $\pi : L \to 2^{\mathscr{P}}$ maps each program location to a precision for formulas. The *boolean predicate abstraction of a formula* φ wrt. precision PS, denoted by $(\varphi)^{PS}$, is the strongest boolean combination of predicates from PS such that $(\varphi)^{PS}$ is implied by φ. Boolean predicate abstractions of region formulae of a program form the *abstract data states* of a predicate analysis. Here, we assume this abstraction to be computed by some standard approach like those given in [16,3,21].

3 Block-Abstraction Memoization

Our technique of block-abstraction memoization (BAM) can be seen as one instance of a *configurable program analysis with dynamic precision adjustment* (CPA+) [6]. While we only present BAM here, it can easily be used together with other features of CPAchecker like for instance block encodings. We first of all describe the basic algorithm for computing the set of abstract reachable states and then describe the block memoization technique. Our CPA+

Algorithm 1. (Part of) CPA+ algorithm for $BAM = (D, \Pi, \rightsquigarrow, \texttt{stop})$.

Input: an initial abstract state $e_0 \in E$ with precision $\pi_0 \in \Pi$, where E denotes the set of elements of the semi-lattice of D
Output: a set of reachable abstract states

```
1: waitlist := {(e_0, π_0)}; reached := {(e_0, π_0)};
2: while waitlist ≠ ∅ do
3:    pop (e, π) from waitlist;
4:    for each e' with e ⇝^π e' do
5:       if ¬stop(e', {e | (e, ·) ∈ reached}, π) then
6:          waitlist := waitlist ∪ {(e', π)};
7:          reached := reached ∪ {(e', π)};
8: return {e | (e, ·) ∈ reached}
```

$BAM = (D, \Pi, \rightsquigarrow, \texttt{stop})$ is a tuple consisting of an abstract domain D, a set of precisions Π, a transfer relation $\rightsquigarrow$ and a termination check $\texttt{stop}$.

- The *abstract domain* $D = (C, \mathscr{E}, [\![\cdot]\!])$ is defined by a set of concrete states C, a semi-lattice $\mathscr{E} = (E, \top, \sqsubseteq, \sqcup)$, and a concretization function $[\![\cdot]\!] : E \to 2^C$. The semi-lattice $\mathscr{E} = (E, \top, \sqsubseteq, \sqcup)$ consists of a set of abstract states E, a top element $\top \in E$, a partial order $\sqsubseteq \subseteq E \times E$, and a join operation $\sqcup : E \times E \to E$. The concretization function maps abstract states to the respective set of concrete states they represent. For the BAM analysis an abstract state $e \in E$ is a pair (l, ψ)[1], where l is a location and ψ is a predicate formula. We have $[\![(l, \psi)]\!] = \{l\} \times \{c \in \mathscr{C} \mid c \models \psi\}$.
- The *set of precisions* Π determines possible precisions of the abstract domain. In case of BAM we have $\Pi = L \to 2^{\mathscr{P}}$: a location is assigned a set of predicates. In an abstract state (l, ψ) with precision π, the formula ψ is built over $\pi(l)$.
- The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ determines which abstract states $e' \in E$ can be reached from $e \in E$ by some edge $g \in G$ of the CFA wrt. precision $\pi \in \Pi$. This is the part where our block memoization technique comes into play. We will describe it below when the basic algorithm is clear.
- The *termination check* $\texttt{stop} : E \times 2^E \times \Pi \to \mathbb{B}$ checks whether a given abstract state $e \in E$ with precision $\pi \in \Pi$ is already "covered" by a given set of abstract states $R \in 2^E$. In our analysis it is simply checked whether there is $e' \in R$ such that $e \sqsubseteq e'$.

Algorithm 1 shows the overall algorithm that computes an overapproximation of the set of reachable states of a program by constructing an abstract state space. It takes an initial abstract state e_0 and an initial precision π_0 as input, maintains a *waitlist* of abstract states not explored yet, and a set *reached* of

[1] In the implementation, an abstract state is a quadruple containing additional information for e.g. the block encodings of CPAchecker, which we however omit here.

already computed abstract states[2]. The algorithm also implicitly constructs the *abstract reachability tree* (ART) referred to in the introduction: the abstract states with precisions in REACHED are the nodes of the ART and we have an edge between two nodes (e, π) and (e', π) if $e \overset{\pi}{\rightsquigarrow} e'$. We write $BAM_A(e_0, \pi_0)$ to describe the set of states returned by this algorithm when started on a CFA A. We say a location l is reachable in the abstraction of a program $P = (A, l_0, l_E)$ wrt. initial precision π_0, if there is some abstract state $e = (l, \cdot)$ such that $e \in BAM_A((l_0, true), \pi_0)$. The lazy abstraction and abstraction refinement technique used in CPACHECKER and also our own approach is not shown here (see Section 4 for a treatment of lazy abstraction).

The interesting part is now the definition of the transfer relation $\rightsquigarrow$. The basic idea of BAM is to consider *blocks* as whole entities for which we separately compute abstract reachability trees which we store and re-use whenever we get to the same block again. Consider again program `NESTED` of Figure 1. If we construct the ART for this program using the predicates $\{i = 0, i = 1, i = 2, j = 0, j = 1, j = 2, k = 0, k = 1, k = 2\}$ we see that e.g. location l_5 is visited four times during construction of the ART. At each visit the abstract states only differ in the valuation of the outer loop variables i and j. Instead of computing the abstraction of the innermost block all over again, we would ideally like to do it only once, cache the obtained results, and re-use it when we visit l_5 next time.

Thus the transfer relation needs to distinguish two cases. The first, standard case occurs when the abstract state $e = (l, \psi)$ does *not* represent the start of a block. In this case, we compute the abstract successor $e' = (l', \psi')$ wrt. an edge $g = (l, op, l')$ and precision π as $\psi' = (SP_{op}(\psi))^{\pi(l')}$, i.e., the boolean abstraction of the strongest postcondition of ψ wrt. operation *op*. In the other case, we have to compute an ART for the block or use a cached block ART. To see how this works we first of all need to define blocks. Intuitively, a block B is a (connected) subgraph of the CFA.

Definition 1. *A* block $B = (L', G')$ *of a CFA* $A = (L, G)$ *consists of a set of locations* $L' \subseteq L$ *and a set of associated edges* $G' = \{(l_1, op, l_2) \in G \mid l_1, l_2 \in L'\}$ *such that for all* $l_1, l_2 \in L'$ *there is a path from* l_1 *to* l_2 *or* l_2 *to* l_1 *formed by edges in* G' *only.*
A block B *has a set of* input locations $In(B) := \{l \mid l \in L' \wedge \exists (l_{prev}, op, l) \in G : l_{prev} \notin L'\}$ *and a set of* return locations $Out(B) := \{l \mid l \in L' \wedge \exists (l, op, l_{succ}) \in G : l_{succ} \notin L'\} \cup \{l_E \mid l_E \in L'\}$.

When partitioning the CFA A into blocks, we require that the partition is *nested* or disjoint, meaning that for blocks $B = (L, G)$, $B' = (L', G')$ we have $L \cap L' = \emptyset$, $L \subset L'$, or $L' \subset L$. Besides this, we do not impose any restrictions for the user-defined partition of the CFA into blocks. In Figure 1 we see a partition of the control flow automaton into three blocks, the inner two comprising the two inner

[2] This algorithm is taken from [6], but we have elided all those parts referring to the different adjustment options and just have taken the part of interest for BAM.

loops, the outer one the whole program. When the dotted lines cut across nodes, these are part of the block.

Given a partition into blocks, we need to describe how the abstract reachability trees for the blocks are computed. Basically, we use the same algorithm for blocks as for the whole CFA, however, we might change (weaken) the precision when computing block ARTs. This is useful because (a) a block might not use some of the current predicates at all, and (b) it enhances the possibilities for re-use when the same block occurs in different subgraphs of the ART with different precisions. Assume for an abstract state $e = (l, \psi)$ with precision π that e is the start of a block. Instead of computing the block ART with respect to $\pi(l)$, we now only compute it for the *relevant predicates* of the block B, denoted by $Preds(B, \pi(l))$. Intuitively, it contains only those predicates of $\pi(l)$ which might be changed or referred to in the block. Analyzing a block only wrt. a set of relevant predicates involves two operations. First, we have to *reduce* (i.e., weaken) the abstract state e's predicate formula to the relevant predicates. Second, we have to *expand* (i.e., strengthen) the resulting abstract states such that then all predicates are taken into account again.

As an example consider the inner most block of the CFA of program `NESTED`. Its input location is l_5 and its return location is l_8. Assume that we reach location l_5 during the analysis in an abstract state $e = (l_5, (i = 0 \wedge j = 0 \wedge k = 0))$ while working with the set of predicates $PS = \{i = 0, i = 1, i = 2, j = 0, j = 1, j = 2, k = 0, k = 1, k = 2\}$ (possibly after many refinements). The relevant predicates for the innermost block are $\{k = 0, k = 1, k = 2\}$ only (the inner loop does not refer to i nor j). Thus we can weaken the predicate formula in e to $k = 0$ (by existentially quantifying over all irrelevant predicates). The analysis (algorithm BAM) is started for the block with this reduced abstract state and gives us an abstract state $(l_8, k = 2)$ for the return location. The predicate formula in here now needs to be expanded: we need to take the conjunction of $k = 2$ with the part of the old predicate that we removed during reduction, i.e. $i = 0 \wedge j = 0$. Again the latter is obtained by existential quantification, this time over the relevant predicates of the block $((\exists (k = 0) : i = 0 \wedge j = 0 \wedge k = 0) \equiv (i = 0 \wedge j = 0))$.

Definition 2. *Let $e = (l, \psi)$ be an abstract state, l an input location of a block $B = (L', G')$, π a precision. Let $Preds(B, \pi(l)) = \{q_1, \ldots, q_m\}$ the set of relevant and $\pi(l) \setminus Preds(B, \pi(l)) = \{p_1, \ldots, p_n\}$ the set of irrelevant predicates of the block B wrt. $\pi(l)$. Then*

$$reduce(e, \{p_1, \ldots, p_n\}) := (l, \exists p_1, \ldots, p_n : \psi) \ .$$

Let furthermore E be some set of abstract states reached by analyzing block B. Then

$$expand(E, e, \{q_1, \ldots q_m\}) := \{(l', \varphi') \mid (l', \varphi) \in E \wedge \varphi' = (\exists q_1, \ldots, q_m : \psi) \wedge \varphi\} \ .$$

With these definitions at hand we can define the transfer relation for the case when the abstract state $e = (l, \psi)$ is an input location of a block $B = (L', G')$. Assume that we are given an edge $g = (l, op, l')$ i.e., $l \in In(B)$, and we have

precision π. Let $\pi' := \{l' \mapsto PS' \mid l' \in L' \wedge PS' = Preds(B, \pi(l'))\}$ be the precision π reduced to the relevant predicates of B. Then the successor states of e are all those states $e' = (l', \psi')$ with $l' \in Out(B)$ and

$$e' \in expand(BAM_B(reduce(e, \pi(l) \setminus Preds(B, \pi(l))), \pi'), e, Preds(B, \pi(l)))$$

Although not explicitly modeled, we actually expect the recursive calls to BAM to be *cached* (*block caches*). In the cache we store the analysis results of blocks for reduced abstract states and precisions. Hence, if we notice that we have already done the same analysis for the block before, we do not recompute it but reuse the cached result. The fact that we first of all reduce the predicate formula allows us to re-use results even if the abstract states differ. This is key to the performance improvement achieved by BAM.

Assuming soundness of the computation of relevant predicates our BAM analysis is also sound. A computation of relevant predicates is said to be sound if the interpretation of predicates classified as irrelevant for some block B cannot change due to an execution of B (otherwise *expand* could introduce unsound information).

Definition 3. *A* computation of relevant predicates $Preds(\cdot, \cdot)$ *is* sound *iff for every block B, every set of predicates $PS \subseteq \mathscr{P}$, and all $p \in PS \setminus Preds(B, PS)$ we have*

$$\forall c \in In(B) \times \mathscr{C} : \forall c' \in Reach_B(\{c\}) : c \models p \iff c' \models p.$$

We can now show that our analysis indeed detects all reachable error states (proof not given due to lack of space).

Theorem 1. *Let $P = (A, l_0, l_E)$ be a program with CFA $A = (L, G)$ that is partitioned into blocks. Let $Preds(\cdot, \cdot)$ be a sound computation of relevant predicates.*
If the error location l_E is reachable in P then l_E is also reachable in every abstraction of P.

As we see here, the analysis is sound for any precision that we might start with, as long as the computation of relevant predicates is sound. Yet, in practice, we do not only want the computation of relevant predicates to be sound but also to be *precise*. That is, the interpretation of predicates classified as irrelevant for some block B should not alter the control-flow of B. If we for example consider a predicate p that occurs in an if-condition as irrelevant, an analysis of the block will explore both branches of the if-clause, whereas a precise computation of relevant predicates will consider p as relevant and thus will possibly only analyze one of the branches of the if-clause.

4 Lazy Abstraction

Lazy abstraction comes into play when applying refinements. Refinements need to be applied when the precision used for the analysis was too coarse to prove

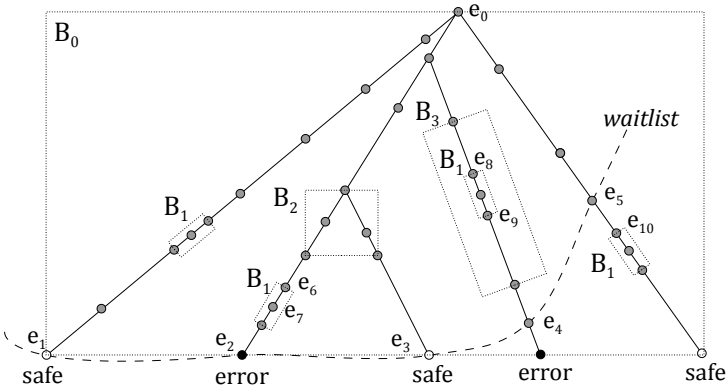


Fig. 2. Example ART as constructed by Algorithm 1

the absence of error states, i.e., when spurious counter examples are found. As running example for this section, consider the ART shown in Figure 2. In the figure, grey circles denote arbitrary abstract states, black circles denote abstract states in the error location, and white circles denote abstract states in a safe location (i.e., locations from which the error location is clearly not reachable anymore). Blocks are indicated by dotted rectangles. The dashed WAITLIST line in the figure indicates which abstract elements are currently in the waitlist of the BAM algorithms (Algorithm 1). In this case, we stopped the algorithm(s), because an error state, namely e_2, was found.

The refinement process now works as described in Algorithm 2. First of all, a counter example (path from e_0 to e_2 in our example) is constructed (line 1) and checked for spuriousness (line 2). These are standard procedures which the model checker performs. If the counter example is not spurious, we found a valid counter example and can report this to the user (line 3). Otherwise, if the counter example is spurious, we can conclude that our analysis was too imprecise and we have thus have to *refine* it such that the spurious counter example is ruled out (line 4-14). To this end, in case of predicate abstraction, we have to find additional predicates. Which predicates are actually needed to rule out the spurious counter example can be determined by e.g. Craig interpolation [18]. In case of lazy refinement, these newly found predicates need to be added to the subtree of the ART where the spurious counter example was found in. For example, in our figure the counter example analysis may report that we shall rebuild the subtree starting with element e_7 and using some new precision π_{new} to rule out the previously found counter example.

For our algorithm we consider the counter example analysis as a black box that returns a "cut" element e_{cut} (with its surrounding block B_{cut} and corresponding precision π_{cut}) from which on we shall rebuild the ART with an also supplied new precision π_{new} (line 5). Note that in case of BAM we actually do not have a single overall ART but rather one ART for each distinct block, abstract element, and precision combination. Thus, to rebuild the subtree starting with e_{cut}, we have to remove e_{cut} in the ART corresponding to $(B_{cut}, e_{cut}, \pi_{cut})$ (line 9-10) and

Algorithm 2. Lazy refinement algorithm

Input: an error element e_{err} in some block B_{err} with precision π_{err}, a cache *cache* : $\mathscr{B} \times E \times \Pi \to ART$, where $\mathscr{B}$ is the set of blocks,
a waitlist for each ART *waitlist* : $ART \to 2^{E \times \Pi}$

Output: $ce := ((B_0, e_0, \pi_0), \ldots, (B_{err}, e_{err}, \pi_{err}))$ if a real counter example was found, updated *waitlist* and ARTs otherwise.

1: $ce :=$ find_path$((B_{err}, e_{err}, \pi_{err}), cache)$;
2: **if** *ce* is not spurious **then**
3: **return** ce; *//Real counter example found*
4: *//Analysis was too imprecise; adapt precision*
5: $((B_{cut}, e_{cut}, \pi_{cut}), \pi_{new}) =$ new_precision(ce);
6: $(B, e, \pi) := (B_{cut}, e_{cut}, \pi_{cut})$;
7: **while** $B \neq nil$ **do**
8: *//outermost block not yet left*
9: $art := cache(B, e, \pi)$;
10: remove_subtree(art, e); *//inplace remove subtree starting with e*
11: $waitlist(art) := waitlist(art) \cup \{(e, \pi_{new})\}$; *//rebuild subtree with new precision*
12: $(B, e, \pi) := last\_caller(B, ce, e_{cut})$; *//node where B was called last before* e_{cut}
13: $\pi_{new} := \pi$; *//precision of outer blocks kept*
14: **return** *waitlist*; *//ARTs got altered inplace*

re-add the element with the new precision π_{new} to the waitlist corresponding to the ART (line 11). Moreover, we have to make sure that the block edge that was used in the outer block gets updated. To this end, we also remove the calling abstract element from the next outer block in a loop till the outermost block was reached (line 7-13).

Referring to our running example, we remove the subtree starting with $e_{cut} := e_7$ in the ART for block $B_{cut} := B_1$ and add (e_7, π_{new}) to the waitlist (for this ART). Afterwards, to update the block edge used at e_6, we also remove the subtree starting with e_6 from the ART for block B_0 and add e_6 with its corresponding precision to the waitlist again.

This concludes the basic refinement algorithm. We yet want to emphasize two important design decisions that turned out to be most efficient in our benchmarks and that are reflected in the algorithm. First, whenever we have to rebuild a cached ART of a block B we do so *without* updating other occurrences of the same cached ART in other parts of outer ARTs. For example, assume that the very same cached ART for B_1 is used for all occurrences of B_1 in Figure 2. Then, when rebuilding the subtree starting with, e.g., e_7, we modify the one and only cached ART for B_1, but only update the block edge used at e_6. This makes sense as we thereby avoid looking into subtrees again that were already proven to be safe (e.g., the branch to e_1). Second, we refine the ARTs of blocks *in-place*, i.e., without keeping a copy of the non-refined ART. This may leave "holes" in the ART. For example, after the refinement of block B_1 there is no ART cached anymore that would explain the (block) edge from e_8 to e_9 in the ART of B_3. If we are lucky, we will not need this information anymore during the further

analysis, namely if we find out that the subtree is safe. Otherwise, if we find the error location in the subtree, we need to reconstruct the information (see below).

To complete the refinement process, we are left to treat two special cases. First, when constructing a counter example to the given error location, we may encounter the "holes" mentioned above. We handle this special case by simply rebuilding the subtree starting with (B_k, e_k, π_k) as done for (B, e, π) in lines 6-13 in Algorithm 2, where (B_k, e_k, π_k) is the last element of the counter example that could be constructed without running into "holes". This forces the CPA algorithm to use the refined ART for the block, which may already rule out the error. If this is not the case, we further refine the refined ARTs as described in Algorithm 2. Thus, in our example, when resuming the BAM algorithm and finding the error state below e_4, we rebuild the subtree starting with e_8. Since the cache at this point only contains a refined ART for block B_1, this refined ART will also be used during rebuilding the subtree. Note that the same holds for the very right branch: when the algorithm proceeds with this branch it will use the refined ART for block B_1 at abstract element e_{10}.

The second special case that needs to be considered in the refinement process may be caused by precision loss of BAM in presence of non-cube abstraction: if, for example, we have a block consisting only of the assignment $b = b * 3$ and an initial non-cube abstraction $(a = 1 \wedge b = 1) \vee (a = 2 \wedge b = 2)$, our BAM algorithm might identify a as irrelevant for the block and reduce the abstraction to $b = 1 \vee b = 2$. Executing the only block statement will lead to an abstraction $b = 3 \vee b = 6$ which gets expanded to $(b = 3 \vee b = 6) \wedge (a = 1 \vee a = 2)$. While this abstraction is sound, it is also clearly less precise than $(a = 1 \wedge b = 3) \vee (a = 2 \wedge b = 6)$, which we get when analyzing the block directly, without using BAM. This impreciseness has be considered in the refinement process, as it may cause the process to fail to rule out certain spurious counter examples. We treat this special case as follows. First of all, we can detect it by simply memorizing the last handled counter example. If we encounter the same counter example during refinement, we apparently failed to rule it out. Second, to avoid running into the same counter example for a third time, we alter the $Preds(B, \cdot)$ function for each block B in the counter example in such a way that it considers *all* predicates occurring in the abstractions of the respective abstract elements of the respective block in the counter example as relevant for the future. This effectively disables BAM for this counter example (*reduce* and *expand* will have no effect) and thus guarantees that the same counter example cannot be encountered again.

5 Experimental Results

We implemented our approach in the program-analysis tool CPAchecker [10] based on the Adjustable-Block Encoding (ABE) [7] predicate analysis. To determine the set of relevant predicates of a block, our implementation currently considers those predicates as relevant that reference a variable occurring in the block (no aliasing assumed). For the partition of the CFA into blocks, we consider (automatically detected) loop bodies and function bodies as blocks.

Table 1. Benchmark results of CPAchecker with block-abstraction memoization in comparison to classic CPAchecker with large-block encoding. Best runtime results are written in bold letters.

Program	CPAchecker with LBE					CPAchecker with BAM				
	#ref. steps	Mem. max	Runtime total	MC	CEA	#ref. steps	Mem. max	Runtime total	MC	CEA
cdaudio_simpl1_BUG	544	632m	17.5	11.2	4.27	122	287m	**10.1**	7.07	1.52
floppy_simpl3_BUG	100	140m	5.37	3.34	1.01	55	89m	**3.95**	2.03	0.93
floppy_simpl4_BUG	225	255m	7.82	4.91	1.63	84	135m	**5.23**	2.95	1.20
kbfiltr_simpl2_BUG	123	67m	3.97	2.15	0.77	30	46m	**2.49**	1.08	0.46
cdaudio_simpl1	555	633m	17.7	11.3	4.44	113	270m	**9.21**	6.10	1.43
diskperf_simpl1	269	537m	14.0	8.59	4.29	101	171m	**7.14**	4.53	1.40
floppy_simpl3	121	153m	6.47	4.11	1.32	66	132m	**5.81**	3.45	1.38
floppy_simpl4	294	405m	9.27	5.90	2.09	101	211m	**7.44**	4.42	1.50
kbfiltr_simpl1	30	37m	2.09	0.97	0.25	14	31m	**1.76**	0.55	0.22
kbfiltr_simpl2	111	90m	3.81	2.07	0.67	26	34m	**2.46**	1.08	0.40
cdaudio.BUG	406	672m	19.6	12.6	3.93	93	259m	**10.3**	5.63	1.85
diskperf.BUG	190	457m	15.7	9.63	4.45	73	225m	**6.40**	3.36	1.62
floppy.BUG	390	297m	14.2	7.81	4.18	97	237m	**8.02**	3.76	2.06
kbfiltr.BUG	121	118m	4.96	2.47	1.10	30	64m	**3.41**	1.34	0.74
parport.BUG	2253	998m	120	61.3	20.7	189	552m	**15.8**	7.97	4.64
cdaudio	610	680m	27.5	17.6	6.43	110	348m	**14.0**	7.89	2.61
diskperf	246	490m	19.2	11.5	5.61	86	284m	**7.91**	4.75	1.76
floppy	785	334m	27.0	14.2	9.75	124	300m	**11.0**	6.18	2.31
parport	2634	1071m	158	90.6	25.0	255	751m	**29.3**	18.8	6.24
mgpd-bluetooth-btmrvl	34	1028m	27.1	23.8	1.16	18	153m	**6.96**	3.94	0.70
mgpd-gpu-drm-i915-i915	20	1468m	51.4	39.4	1.53	9	369m	**14.3**	1.28	1.28
mgpd-net-pppox	8	31m	**2.38**	0.75	0.14	13	42m	2.76	0.93	0.43
mgpd-atm-eni	*out of memory*					13	479m	**9.97**	5.91	0.48
mgpd-block-drbd-drbd	*out of memory*					6	207m	**12.6**	0.99	0.26
mgpd-block-paride-pt	*out of memory*					12	378m	**10.5**	2.60	6.21
mgpd-hwmon-it87	*out of memory*					3	107m	**4.84**	1.87	0.10
mgpd-net-atl1c-atl1c	*out of memory*					11	492m	**13.2**	8.27	0.67
mgpd-net-sis900	*out of memory*					24	378m	**10.9**	6.86	0.65
mgpd-scsi-megaraid	*out of memory*					11	539m	**14.9**	9.07	0.34
mgpd-staging-et131x-et131x	*out of memory*					4	432m	**11.9**	5.85	0.18
mgpd-video-aty-aty128fb	*out of memory*					28	1461m	**32.3**	3.24	26.9
nested_1_6	8	22m	**1.14**	0.19	0.04	8	18m	1.31	0.24	0.05
nested_2_6	14	33m	2.32	1.24	0.16	14	34m	**1.72**	0.53	0.26
nested_3_6	22	129m	11.8	8.94	2.02	29	35m	**5.10**	1.00	3.15
nested_4_6	30	107m	216	129	86.4	85	130m	**147**	1.95	144
nested_5_6	*out of memory*					*timeout (>15min)*				

Using this configuration we submitted our implementation to the 1st Intl. Competition on Software Verification (held at TACAS 2012). With a total score of 280 points we won the competition in the category "Overall" [26]. In the following we report on some results of our implementation when compared to CPAchecker without BAM on benchmark programs of the competition as well as on our constructed `NESTED` programs (nested_n_m for n nested loops each counting to m). Please refer to [8] for a comparison with other verification tools. All experiments were performed on a 64bit Ubuntu 11.04 machine with 4GB RAM, an Intel i5 CPU with 2.53GHz, and OpenJDK 6 64-Bit Server VM (build 20.0-b11, IcedTea6 1.10.2)[3]. Table 1 shows an excerpt of the results of our benchmarks.

In the table, we provide the number of refinements needed, the maximum amount of heap space occupied, and the total runtime of the verification algorithm when using CPAchecker with or without BAM (in both cases we configure ABE as Large-Block Encoding (LBE) [5], i.e., to compute abstractions at loops and functions). Moreover, we further partition the total runtime into time spent for model checking (MC) and for counter example analysis (CEA; this includes building the counter example, determining whether it is spurious or not, and computing interpolants to receive additional predicates that rule out the spurious counter example). All runtimes are given in seconds.

As expected, the time spent on model checking is reduced dramatically for the `NESTED` programs. With only four nested loops we already achieve a speed-up in model checking of more than factor 50 by using BAM. Note that the length of the counter examples grows exponential with the number of nested loops. Since the predicate analysis with BAM has to construct these counter examples as well during refinement, it also suffers from an exponential blowup, leading overall speed-up of less than factor two.

More interestingly, using BAM for the NT-drivers (first two blocks in the table) we gain an overall speed-up of about factor three. Clearly, the success of the technique relies strongly on the structure of the given programs. Hence, on some examples the overall speed-up is even about factor ten, while on other example the speed-up is almost non-existent. Yet, the technique never imposes any significant overhead. Also it can be noticed that even the memory consumption is lower in average when using BAM. This reduction in memory consumption can be explained by the reduced size of the ART.

Finally, the third block of programs in Table 1 shows the result for a set of 64bit Linux device drivers. As it turns out, for this benchmark set, BAM can drastically improve the efficiency of CPAchecker. That is, with BAM all nine programs, for which the analysis failed without BAM, can now be efficiently verified as safe. This set of examples therefore shows that due to its increase in efficiency BAM allows to successfully analyze programs that were not possible to analyze without.

[3] BAM is fully incorporated into CPAchecker. To rerun the experiments just download the source code of CPAchecker 1.1 (`http://cpachecker.sosy-lab.org/`) and run CPAchecker configured with "predicateAnalysis-lbe" and "predicateAnalysis-abm", respectively, on the benchmark programs.

This is also reflected on the complete benchmark set of the software verification competition: with BAM CPAchecker is able to correctly analyze 211 of 277 programs, whereas without BAM (and using LBE) CPAchecker can only correctly analyze 167 of 277 programs[4].

6 Conclusion

In this paper we presented block-abstraction memoization, a modular approach for model checking (predicate) abstract state spaces. The approach relies on a user-defined adjustment of blocks for the modularization of the analysis. For each block a set of relevant predicates is computed to speed-up the analysis of the blocks and to efficiently cache intermediate analysis results. We implemented our technique into the program verification tool CPAchecker and showed on basis of a given benchmark set that our technique indeed speeds up the verification task.

Related Work. As already indicated in the introduction, the idea of block summarizations is not new. Summarizations are used for the analysis of heap structures [2,14], finite state machine specifications [12], and program termination [25]. In the following we compare our work to those approaches which we consider conceptually closest to ours.

The work closest to our technique is the paper of Ball and Rajamani [4]. In the paper the underlying technique of the boolean program model checker Bebop is described. Bebop is part of the SLAM framework [1], which implements an eager (i.e. non-lazy) counter example guided (predicate) abstraction refinement cycle for C-programs. In [4] the interprocedural dataflow algorithm of Reps et al. [24] is generalized and transferred to boolean programs. This involves the use of *summarizations* that reflect the in- and output behavior of functions. However, this technique is only used in an eager framework and for functions only.

Adjustable-Block Encoding [7] is a technique that allows a predicate analysis to compute predicate abstractions only at user-defined locations. Large-Block Encoding [5] is a special case of ABE that computes abstractions only at loop heads and function calls. In [5] LBE is shown to be vastly more efficient than Single-Block Encoding, i.e., the computation of predicate abstractions after each operation. Basically, ABE and LBE are orthogonal concepts to BAM and can thus, in principle, be freely combined with BAM. As we however only define the reduce and expand operation on abstract elements with a predicate abstraction, we currently require ABE to be configured such that an abstraction is computed at the start and end of each block (as defined by BAM). In practice, it furthermore has turned out that configuring ABE to compute an abstraction *only* at the start and end of a block yields the best performance.

The approach of the tool SLAB [11] will in some cases also consider blocks or functions as entities of which no further detailed analysis is necessary. Instead of

[4] Measured using a scoring schema as defined by the Competition on Software Verification 2012 CPAchecker with BAM yields 282 points on our machine, compared to 206 points when only using LBE.

starting with the CFA of a program, [11] starts with a very coarse abstraction only distinguishing initial and error states. Nodes for particular locations are only introduced when needed, and thus locations inside functions might not be explicitly represented. However, there is no notion of re-use of already explored parts, blocks occurring several times will also be analyzed several times.

In [15] Godefroid et al. describe a verification approach that is based on the combination of a may and must analysis, using a SLAB alike predicate abstraction for the may analysis and dynamic test generation for the must analysis. They use function summaries in both analyses and show how the analyses complement each other to yield better function summarizations. In their approach function summaries are however immutable and not subject to any refinement. Thus, although employed in a lazy framework, Godefroid et al. did not face similar challenges in the refinement process as we discussed in Section 4.

Future Work. For the future we want to look at more sophisticated static analyses for the computation of relevant predicates of a block. This is especially required for programs with pointer-aliasing. Also, we want to look for other block partitions than just loop bodies and function bodies. For example, it may be beneficial to skip definitions like `localVar := someGlobalVar` at the beginning of blocks for loop and function bodies, because this may reduce the amount of relevant predicates of the block and thereby lead to more cache hits.

Looking on how we defined BAM in the CPAchecker framework, we also believe that the technique can, in principle, be generalized to work on top of arbitrary CPAs like e.g. explicit state analysis or shape analysis [6]. Such a generalization of the technique would allow us to combine different CPAs, as done in CPAchecker, while still benefiting from the modular BAM approach.

Finally, an interesting application of our technique is proof-carrying code ([22]). The idea here is that BAM can be used to construct more compact proofs for programs than with classical (lazy) predicate analysis as done in [17].

Acknowledgement. We would like to thank Philipp Wendler for his extensive help with the integration of BAM into CPAchecker.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
2. Ball, T., Hackett, B., Lahiri, S.K., Qadeer, S., Vanegue, J.: Towards Scalable Modular Checking of User-Defined Properties. In: Leavens, G.T., O'Hearn, P., Rajamani, S.K. (eds.) VSTTE 2010. LNCS, vol. 6217, pp. 1–24. Springer, Heidelberg (2010)
3. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
4. Ball, T., Rajamani, S.: Bebop: A Symbolic Model Checker for Boolean Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (2000)

5. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD 2009, pp. 25–32. IEEE (2009)
6. Beyer, D., Henzinger, T., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: ASE 2008, pp. 29–38. IEEE Computer Society (2008)
7. Beyer, D., Keremoglu, M., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: FMCAD 2010, pp. 189–197 (2010)
8. Beyer, D.: Competition on Software Verification. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 504–524. Springer, Heidelberg (2012)
9. Beyer, D., Henzinger, T., Jhala, R., Majumdar, R.: The software model checker BLAST. STTT 9, 505–525 (2007)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
11. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. Fundam. Inform. 89(4), 369–392 (2008)
12. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. IEEE Transactions on Software Engineering 30(6), 388–402 (2004)
13. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
14. Dillig, I., Dillig, T., Aiken, A., Sagiv, M.: Precise and compact modular procedure summaries for heap manipulating programs. SIGPLAN Not. 46, 567–577 (2011)
15. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional must program analysis: unleashing the power of alternation. SIGPLAN Not. 45, 43–56 (2010)
16. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
17. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-Safety Proofs for Systems Code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
18. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) POPL 2004, pp. 232–244. ACM (2004)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL 2002, pp. 58–70 (2002)
20. Ivancic, F., Yang, Z., Ganai, M.K., Gupta, A., Ashar, P.: Efficient SAT-based Bounded Model Checking for Software Verification. Theoretical Computer Science 404, 256–274 (2008)
21. Lahiri, S.K., Nieuwenhuis, R., Oliveras, A.: SMT Techniques for Fast Predicate Abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 424–437. Springer, Heidelberg (2006)
22. Necula, G.C.: Proof-carrying code. In: POPL 1997, pp. 106–119. ACM, New York (1997)
23. Podelski, A., Rybalchenko, A.: ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
24. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. In: POPL 1995, pp. 49–61. ACM, New York (1995)
25. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop Summarization and Termination Analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)
26. Wonisch, D.: Block Abstraction Memoization for CPAchecker (Competition Contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 531–533. Springer, Heidelberg (2012)

Heuristic-Guided Abstraction Refinement for Concurrent Systems

Nils Timm, Heike Wehrheim, and Mike Czech

Department of Computer Science, University of Paderborn, Germany
{nils.timm,wehrheim,mczech}@mail.upb.de

Abstract. Predicate abstraction is an established technique in software verification. It inherently includes an abstraction refinement loop successively adding predicates until the right level of abstraction is found. For concurrent systems, predicate abstraction can be combined with *spotlight abstraction*, further reducing the state space by abstracting away certain processes. Refinement then has to decide whether to add a new predicate or a new process. Selecting the right predicates and processes is a crucial task: The positive effect of abstraction may be compromised by unfavourable refinement decisions. Here we present a *heuristic* approach to abstraction refinement. The basis for a decision is a set of *refinement candidates*, derived by *multiple counterexample-generation*. Candidates are evaluated with respect to their *influence* on other components in the system. Experimental results show that our technique can significantly speed up verification as compared to a naive abstraction refinement.

1 Introduction

Predicate abstraction [1] is one of the most promising techniques for reducing the complexity of software model checking. It proceeds by generating an *abstract* state space of the system to be analysed. In this abstraction, concrete states of the system are mapped to abstract states over a finite set of predicates. If the abstraction turns out to be too coarse for verification, *abstraction refinement* incrementally adds new predicates in order to arrive at a level of abstraction where the property of interest can be proven. Abstraction refinement is typically based on the extraction of new predicates from spurious counterexamples (error paths that are only feasible in the abstraction, but not in the concrete system) – either by computing weakest preconditions [4,2] or by interpolation [11].

For concurrent systems composed of many processes, predicate abstraction can be combined with *spotlight abstraction* [18,14]. Here a so-called *spotlight* is set on certain processes, whereas the rest of the system is kept in the *shade*. Now, classical predicate abstraction is applied to the processes in the spotlight, while the shade processes are summarised into one approximative process, i.e. they are nearly completely abstracted away. This generally enables a verification of concurrent systems on very small abstractions, in particular when the property to be checked is local to some processes. However, such abstractions may also be too coarse for a definite result in verification, and thus, may necessitate refinement.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 348–363, 2012.

The spotlight principle adds a new facet to iterative abstraction refinement. In every refinement step a *new predicate* can be added to the abstraction *or* an *additional process* can be drawn from the shade into the spotlight. Furthermore, it remains to decide which *particular* predicate or process to select. The selection of the 'right' processes and predicates for refinement is the crucial part of the overall procedure. Naive decisions, even when guided by spurious counterexamples, often lead to an unnecessary blow-up of the state space. Typically, a spurious counterexample hints to a large set of processes and predicates that are *potentially* relevant for a definite result in verification. Hence, adding the *whole set* to the abstraction may introduce a considerable amount of redundancy or may even cause the model checker to run out of memory. On the contrary, selecting *one arbitrary* (e.g. the first discovered) *element* may guide the refinement in unfavourable directions. An example would be the complete unrolling of a loop by iteratively adding all possible predicates over the loop variable – though adding a certain process (possibly affecting the loop variable) might immediately reveal a definite answer in verification.

In this paper, we introduce a *heuristic-guided* abstraction refinement framework for verifying concurrent systems. The idea of using heuristics in model checking has already been explored in *directed model checking* [6,12]. Therein, the exploration of the state space is based on heuristic search strategies in order to find *counterexamples of minimal length*. In contrast, we aim at *minimising the size of the final abstraction* on which a definite result in verification can be obtained. The use of heuristics for the purpose of abstraction refinement has also been employed in [9,10] for the verification of *hardware designs*, however, not building on a predicate abstraction framework but on abstractions making sets of input variables invisible.

Our approach uses predicate abstraction combined with spotlight abstraction and is based on *three-valued* model checking [15]. Hence, the outcome of a verification run can be *true*, *false* or *unknown*. Definite results can be directly transferred to the original system; only *unknown* necessitates abstraction refinement. In the latter case, our model checker returns an *unconfirmed counterexample* – a potential error path with *unknown* transitions and predicates. These 'unknowns' point to a set of *refinement candidates*: unconsidered predicates and shade processes that might help to resolve some uncertainty in the unconfirmed counterexample. In order to enlarge the set of refinement candidates, we employ *multiple counterexample-generation*. The candidates are then evaluated in terms of their usefulness for guiding abstraction refinement towards a definite result in verification. This heuristic evaluation is based on an *abstraction dependence analysis* which measures the dependencies between each candidate and the rest of the abstraction. We present a heuristic framework for selecting the presumably 'best' candidate for refinement. The fully automatic approach has been implemented on top of our model checking tool 3Spot. Experiments on checking CTL [3] properties (safety and liveness) of concurrent systems show that our heuristic-guided abstraction refinement significantly outperforms naive refinement approaches in both size of the final abstraction and verification time.

2 Basics

We start with a brief introduction to the systems that we consider in our work. A *concurrent system Sys* consists of a fixed number n of non-uniform processes $Proc_1$ to $Proc_n$ composed in parallel: $Sys = \|_{i=1}^{n} Proc_i$. It is defined over a set of variables $Var = Var_s \cup \bigcup_{i=1}^{n} Var_i$ where Var_s is a set of shared variables and $Var_1, \ldots, Var_n$ are sets of local variables associated with the processes $Proc_1, \ldots, Proc_n$, respectively. Variables either have a basic type ($bool$, int) or an array type ($int \to bool$, $int \to int$). The state space of a system corresponds to the set S_{Var} of all possible valuations of the variables. Given a state $s \in S_{Var}$ and an expression e over Var, then $s(e)$ denotes the valuation of e in s.

In our approach, we particularly focus on concurrent systems with asynchronous message passing, also referred to as *message passing systems*. We therefore introduce finite-length communication channels based on circular buffers. Such channels can be modelled using a set of basic variables and an array:

Definition 1. *An* asynchronous communication channel *of length $m \in \mathbb{N}^+$ and type $t \in \{bool, int\}$ is given by a tuple $c = (buffer_c, rear_c, front_c, full_c, empty_c)$ where*

- $buffer_c$: *array* $[m]$ *of* t
  an array representing the channels content,
- $rear_c, front_c$: int
  pointer variables for the rear and front elements,
- $full_c, empty_c$: $bool$
  Boolean variables indicating whether the channel is full or empty.

The initial configuration of a channel c is denoted by the expression

$$rear_c = 0 \wedge front_c = 0 \wedge \neg full_c \wedge empty_c,$$

hence c is initially empty.

As we can see, message passing systems can be straightforwardly incorporated into our general notion of concurrent systems: The set of channel-related variables just corresponds to the set Var_s of shared variables. We assume that in our message passing systems shared variables are solely channel-related, and thus, all other variables are local. Hence, inter-process communication can only be established by sending and receiving messages through channels. As an example, consider the following system:

$$c : \texttt{channel [2] of int}$$

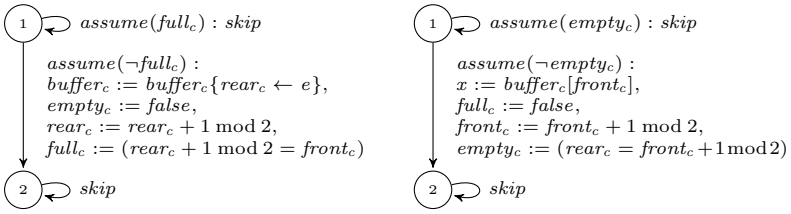
$$Proc_1 :: \begin{bmatrix} 1 : send(c, e) \\ 2 : \texttt{End} \end{bmatrix} \;\Big\|\; Proc_2 :: \begin{bmatrix} 1 : receive(c, x) \\ 2 : \texttt{End} \end{bmatrix}$$

In this message passing system we have a channel c of length 2 and type int. Process $Proc_1$ sends the expression e to the channel whereas $Proc_2$ receives a value from c and stores it in some local variable x. Our channels pass messages in first-in, first-out manner. Moreover, sending to full channels and receiving on empty channels will cause busy waiting.

We formally represent processes $Proc_i$ as control flow graphs (CFGs) $G_i = (Loc_i, \delta_i)$ where Loc_i is the set of control locations and $\delta_i \subseteq Loc_i \times Op \times Loc_i$ is a transition relation labelling edges with operations from a set Op:

Definition 2. *Let* $Var = \{x_1, \ldots, x_{|Var|}\}$ *be a set of variables. The* set of operations Op *on these variables consists of all statements of the form* $assume(e) : x_1 := e_1, \ \ldots, \ x_{|Var|} := e_{|Var|}$ *where* $e, e_1, \ldots, e_{|Var|}$ *are expressions over* Var.

Thus, every operation consists of a guard and a compound assignment. For convenience, we sometimes just write e instead of $assume(e)$. Moreover, we omit the *assume* part completely when the guard is *true*. The CFGs for the processes in Figure 1 – and particularly the semantics of the communication statements $send(c, e)$ and $receive(c, x)$ – are given by



where *skip* denotes the empty assignment.

A concurrent system given by n single control flow graphs $G_1, \ldots, G_n$ can be modelled by one compound CFG $G = (Loc, \delta)$ where $Loc = \bigtimes_{i=1}^{n} Loc_i$. In every combined location $(l_1, \ldots, l_n) \in Loc$ one enabled transition from a process is non-deterministically selected. Transitions are additionally labelled with the identifiers of the associated processes (i.e. $\delta \subseteq Loc \times Op \times [1..n] \times Loc$).

Control flow graphs enable us to model concurrent systems formally. However, for an efficient verification we additionally need to reduce the models' state space complexity. Therefore, we follow an approach based on predicate abstraction. In common approaches (e.g. [1]) the original system is *overapproximated* by a *Boolean abstraction*, and thus, only universal system properties are preserved. This may lead to *spurious* counterexamples in verification. Here, we employ a *three-valued abstraction* [15] with a third truth value *unknown*. Such an abstraction is an approximation in the sense that *all* definite results (*true*, *false*) in verification can be transferred to the original system. Hence, there are no spurious counterexamples. Only *unknown* results necessitate abstraction refinement. In this case a so-called *unconfirmed* counterexample is returned – a potential error path in the abstract system with some 'unknowns'.

In our abstract systems operations do not refer to concrete variables but to predicates over variables. Thus, every concrete operation op is approximated by an abstract operation

$$op_a \ \equiv \ assume(pe) : p_1 := pe_1, \ \ldots, \ p_{|Pred|} := pe_{|Pred|}$$

where $Pred = \{p_1, \ldots, p_{|Pred|}\}$ is a set of atomic predicates over Var, and $pe, pe_1, \ldots, pe_{|Pred|}$ are expressions over $Pred$.

Our abstraction framework is based on a three-valued domain: the valuation of a predicate in a state s can be *true*, *false* or $\perp$, i.e. *unknown*. *Unknown* is in fact a valid truth value as we operate with the *Kleene logic* $\mathcal{K}_3$ [8]. We use $\perp$ to model the loss of information due to abstraction. Logical expressions over *Pred* often take the three-valued form $choice(a, b)$ for Boolean expressions a, b with the following semantics:

$$s\,(choice\,(a,b)) = \begin{cases} true & \text{if } s(a) \text{ is } true \\ false & \text{if } s(b) \text{ is } true \\ \perp & \text{else} \end{cases}$$

This lets us define an approximation relation $\preceq$ on logical expressions. A three-valued expression $pe := choice(a, b)$ over *Pred* approximates a Boolean expression e over *Var* iff a logically implies e and b logically implies $\neg e$. The approximation can be extended to operations as follows:

$$op_a \preceq op \quad \equiv \quad pe \preceq e \ \wedge \ \bigwedge_{i=1}^{|Pred|} pe_i \preceq wp_{op}(p_i)$$

where op and op_a are defined as before and $wp_{op}(p_i)$ is the weakest precondition of the predicate p_i with respect to the operation op. $wp_{op}(p)$ is a predicate expression denoting the set of all states s such that op executed in s results in a state where p holds.

Now, given a concurrent system *Sys* and a set of atomic predicates $Pred = \{p_1, \ldots, p_{|Pred|}\}$, then we can derive an abstract system Sys^a that approximates *Sys* via calculating weakest preconditions and approximating them by three-valued expressions over *Pred*. The original control flow is not affected by predicate abstraction. Hence, an abstract process $Proc^a$ approximates a concrete process *Proc* if they have isomorphic CFGs and the operations in $Proc^a$ approximate the corresponding ones in *Proc*.

As a computational model for our systems we use fair three-valued Kripke structures. A three-valued Kripke structure extends a classical Kripke structure by a three-valued domain for transitions and labellings of states. The logical basis for this is the aforementioned Kleene logic $\mathcal{K}_3$. Furthermore, our Kripke structures are enriched by fairness constraints.

Definition 3. *A* fair three-valued Kripke structure *over a set of atomic predicates AP is a 4-tuple $K = (S, R, L, \mathcal{F})$ where*

- *S is a set of states,*
- *$R : S \times S \to \{true, \perp, false\}$ is a total three-valued transition relation,*
- *$L : S \times AP \to \{true, \perp, false\}$ is a three-valued function labelling states with atomic predicates,*
- *$\mathcal{F} \subseteq \mathcal{P}(R^{-1}(\{true, \perp\}))$ is a set of fairness constraints where each constraint is a set of non-false transitions and $F \in \mathcal{F}$ requires that every fair path takes infinitely often a transition from F.*

A path π of a Kripke structure K is an infinite sequence of states $s_0 s_1 s_2 \ldots$ with $R(s_i, s_{i+1}) \in \{true, \perp\}$; π_i denotes the i-th state of π, Π_s denotes the set of *all*

paths starting in $s \in S$ whereas Π_s^{fair} denotes the set of all *fair* paths starting in $s \in S$.

A concurrent system can be represented as a Kripke structure as follows:

Definition 4. *Let* $Sys = \|_{i=1}^{n} Proc_i$ *be a concurrent system given by a compound CFG* $G = (Loc, \delta)$. *Moreover, let Pred be a set of predicates over the system variables Var. The corresponding Kripke structure is* $K = (S, R, L, \mathcal{F})$ *over* $AP = Pred \cup \{Proc_i @ j \mid i \in [1..n], j \in Loc_i\}$ *with*

- $S := Loc \times S_{Var}$,
- $R(\langle l, s\rangle, \langle l', s'\rangle) :=$
  $\bigvee_{i=1}^{n} \big(\delta(l, op, i, l') \wedge s(e) \wedge s'(v_1) = s(e_1) \wedge \ldots \wedge s'(v_{|Var|}) = s(e_{|Var|})\big)$
  where $op = assume(e) : v_1 := e_1, \ldots, v_{|Var|} := e_{|Var|}$,
- $L(\langle l, s\rangle, p) := s(p)$ *for any* $p \in Pred$,
- $L(\langle l, s\rangle, Proc_i @ j) := \begin{cases} true & if \;\; l_i = j \\ false & else \end{cases}$
  where l_i *is the location of* $Proc_i$ *in the combined location* l,
- $\mathcal{F} := \big\{\{(s, s') \in R_i^{-1}(\{true, \bot\})\}_{i \in [1..n]}\big\}$
  for each process $Proc_i$ *a fairness set* F_i *with all associated transitions.*

Note that the set AP is explicitly extended by propositions of the form $Proc_i @ j$, referring to the processes' program counters. As long as we consider *concrete* systems, the obtained Kripke structures will be actually two-valued. The Kripke structure semantics for *abstract* concurrent systems straightforwardly follow those for concrete systems.

For specifying properties of three-valued Kripke structures we use the computational tree logic (CTL) [3]:

Definition 5. *Let* AP *be a set of atomic predicates and* $p \in AP$. *The* syntax of CTL *is given by*

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid EX\psi \mid AX\psi \mid EF\psi \mid AF\psi \mid EG\psi \mid AG\psi \mid E[\psi U \psi] \mid A[\psi U \psi].$$

The validity of a CTL formula ψ in a state s of a Kripke structure K is denoted by $[K, s \models \psi]$.

Definition 6. *Let* $K = (S, R, L, \mathcal{F})$ *be a three-valued Kripke structure over* AP, $p \in AP$ *and* $\psi \in CTL$. *Then the evaluation of* ψ *in a state* s *of* K, $[K, s \models \psi]$, *is inductively defined as follows*

$$\begin{aligned}
[K, s \models p] &:= L(s, p) \\
[K, s \models \neg\psi] &:= \neg\,[K, s \models \psi] \\
[K, s \models \psi \vee \psi'] &:= [K, s \models \psi] \;\vee\; [K, s \models \psi'] \\
[K, s \models EX\psi] &:= \textstyle\bigvee_{s' \in S} R(s, s') \wedge [K, s' \models \psi] \\
[K, s \models EG\psi] &:= \textstyle\bigvee_{\pi \in \Pi_s^{fair}} \bigwedge_{i \in \mathbb{N}} [K, \tau_i \models \psi] \\
[K, s \models E[\psi U \psi']] &:= \textstyle\bigvee_{\pi \in \Pi_s^{fair}} \bigvee_{i \in \mathbb{N}} \Big([K, \pi_i \models \psi'] \wedge \bigwedge_{0 \le j < i} [K, \pi_j \models \psi]\Big)
\end{aligned}$$

(The remaining CTL operators can be derived by the usual dualities.)

Since we operate in a three-valued domain, the evaluation of a CTL formula on a Kripke structure may yield *true*, *false* or *unknown*. The latter case may occur when the abstraction is too coarse, i.e. the set of predicates is not sufficiently large enough, and thus, some three-valued expressions evaluate to $\bot$. In the next section we will see how the third truth value *unknown* may also arise due to spotlight abstraction.

3 Spotlight Abstraction

Predicate abstraction is a powerful technique in cutting down the state space of systems with large-domain variables. However, in concurrent systems the space complexity furthermore exponentially grows with the number of processes composed in parallel. We therefore combine predicate abstraction with *spotlight abstraction* [14] – a specific abstraction technique for concurrent systems, where a *spotlight* is set on certain processes of interest while the remaining ones are kept in the *shade*. Now, classical predicate abstraction is applied to the processes in the spotlight whereas shade processes are summarised into one approximative component $Proc_{Shade}$. This process neglects the original control flow of the processes in the shade. Instead it approximates operations on shared variables occurring in shade processes by continuously modifying predicates over those variables. Due to the approximative character of $Proc_{Shade}$ and the inherent loss of information about the shade processes, predicates might be set to *unknown*. For illustration, consider the following example:

$$c : \texttt{channel [1] of int}$$

$$Proc_1 :: \begin{bmatrix} \texttt{loop forever} \\ \begin{bmatrix} 1 : send(c, e) \\ 2 : \texttt{Progress} \end{bmatrix} \end{bmatrix} \quad \Big\|_{i=2}^{n} \; Proc_i :: \begin{bmatrix} \texttt{loop forever} \\ \begin{bmatrix} 1 : receive(c, x_i) \\ 2 : \texttt{Progress} \end{bmatrix} \end{bmatrix}$$

In this system we have n processes communicating via channel c. To validate the liveness property $\mathbf{AG}(\mathbf{AF}\ Proc_1@\texttt{Progress})$ (i.e. whether process $Proc_1$ will continuously reach the `Progress` location) it is sufficient to just take $Proc_1$ and $Proc_2$ into the spotlight, and to construct an abstract system over the predicates $empty_c$ and $full_c$. The processes $Proc_3, \ldots, Proc_n$ can be summarised into the following shade component (CFG representation):

$$Proc_{Shade} :: \circlearrowright \; empty_c := choice(false, \neg empty_c), full_c := choice(full_c \vee \neg empty_c, false)$$

Generally, a spotlight abstraction of a concurrent system $Sys = \|_{i=1}^{n}\ Proc_i$ is defined by a set of *spotlight processes* $Spot(Proc) \subseteq Proc$, where $Proc = \{Proc_1, \ldots, Proc_n\}$, and a set of *spotlight predicates* $Spot(Pred) \subseteq Pred$, where $Pred$ is the set of all atomic predicates over the system variables. We refer to the overall spotlight as $Spot = Spot(Proc) \cup Spot(Pred)$. The corresponding shade is characterised by the complementary sets $Shade(Proc) = Proc \setminus Spot(Proc)$ and $Shade(Pred) = Pred \setminus Spot(Pred)$, and moreover, $Shade = Shade(Proc) \cup Shade(Pred)$. Three-valued spotlight abstraction now can be applied to concurrent systems according to the following definition:

Definition 7. *Let* $Sys = \|_{i=1}^{n} Proc_i$ *be a concurrent system and Pred the set of all predicates over the system variables. Moreover, let Spot be a given set of spotlight processes and predicates. Then the abstract system* $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \parallel Proc_{Shade}$ *approximates Sys iff*

- *for every* $Proc_i \in Spot(Proc)$*:* $Proc_i^a$ *approximates* $Proc_i$*,*
- $Proc_{Shade}$ *is a CFG with one location and a single loop labelled with an abstract operation* op_a *over* $Spot(Pred)$ *such that* all *concrete operations op that occur in shade processes are approximated by* op_a*.*

This is the basic definition of three-valued spotlight abstraction, taken with slight changes from [14] (in the original approach every spotlight predicate that is modified in the shade is just set to *unknown* in op_a). For our message passing systems, we sometimes use an enhanced approach to spotlight abstraction: Instead of summarising the shade into a *single* approximative component $Proc_{Shade}$ we introduce *multiple* components $Proc_{Shade}^1, \ldots, Proc_{Shade}^m$, each approximating only a subset of shade processes (e.g. only processes that communicate on a particular channel). In this way, we are able to preserve more concrete behaviour in every shade component and thus, to obtain more definite results in verification.

The following theorem, adapted from our previous work [14], relates the verification results of concrete and abstract systems for *corresponding* states. Given two labelling functions L and L^a over a set of atomic predicates AP, then we say, a concrete state s and an abstract state s^a correspond to each other if both states refer to the same control flow locations and the labelling of s is more definite than the labelling of s^a, i.e. $\forall\, p \in AP : L^a(s^a, p) \leq_{\mathcal{K}_3} L(s, p)$[1].

Theorem 1. *Let* $Sys = \|_{i=1}^{n} Proc_i$ *be a concurrent system, Spot the set of spotlight predicates and processes, and* $Sys^a = \|_{Proc_i \in Spot(Proc)} Proc_i^a \parallel Proc_{Shade}$ *the corresponding abstract system. Moreover, let* $K = (S, R, L, \mathcal{F})$ *be the Kripke structure representing Sys, and* $K^a = (S^a, R^a, L^a, \mathcal{F}^a)$ *the Kripke structure representing* Sys^a*. Then for any two corresponding states* $s \in S$ *and* $s^a \in S^a$ *and for any CTL formula* ψ *over the predicates in Spot or locations of spotlight processes the following holds:*

$$[K^a, s^a \models \psi] \;\leq_{\mathcal{K}_3}\; [K, s \models \psi]$$

Hence, spotlight abstraction enables us to check temporal logic properties of concurrent systems on (usually very small) abstract models. Definite results (*true*, *false*) can be transferred to the original systems. However, due to our three-valued domain we might also obtain an *unknown* result; i.e. the current abstraction might be not precise enough. In the next section we will see how imprecise abstractions can be iteratively refined under heuristic guidance.

4 Heuristic-Guided Refinement

Checking temporal logic properties on three-valued spotlight abstractions might yield the truth value $\perp$, which indicates that it is uncertain whether the desired

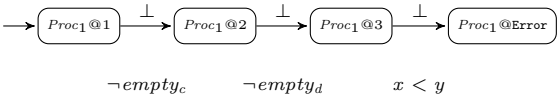
[1] Here '$\leq_{\mathcal{K}_3}$' denotes the *information order* of the Kleene logic $\mathcal{K}_3$ with $\perp \leq_{\mathcal{K}_3}$ *true*, $\perp \leq_{\mathcal{K}_3}$ *false* and *true*, *false* incomparable.

property holds for the original system or not. In this case our model checker additionally returns an *unconfirmed counterexample* – a potential error path with some unknown transitions or predicates. This path provides hints about which *details* of the original system are required for obtaining a definite verification result, but missing in the current abstraction; e.g. assume conditions that are not contained in the set of spotlight predicates, or shade processes that communicate on relevant channels. These missing details indicate possible refinement steps. For illustration we consider the following system where several processes communicate via channel c, but only $Proc_1$ attempts to communicate via d:

$$c, d : \texttt{channel [1] of int}$$

$$Proc_1 :: \begin{bmatrix} 1 : receive(c, x) \\ 2 : receive(d, y) \\ 3 : \texttt{if}(x < y) \\ 4 : \quad \texttt{Error} \\ \quad \texttt{else} \\ 5 : \quad \texttt{End} \end{bmatrix} \quad \Big|\Big|_{i=2}^{n} Proc_i :: \begin{bmatrix} \dots \\ send(c, e_i) \\ \dots \end{bmatrix} \quad \Big|\Big|_{j=n+1}^{m} Proc_j :: \begin{bmatrix} \dots \\ receive(c, z_j) \\ \dots \end{bmatrix}$$

Model checking the safety formula **AG** $\neg(Proc_1@\texttt{Error})$ on the initial abstraction[2] given by $Spot(Proc) = \{Proc_1\}$ and $Spot(Pred) = \varnothing$ yields the following unconfirmed counterexample, annotated with *refinement candidates*:

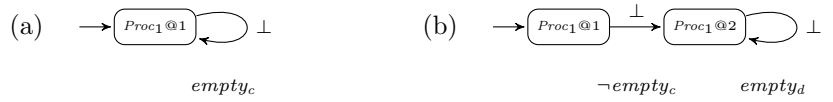


Now, a naive approach to refinement would be to add *all* candidates ($\neg empty_c$, $\neg empty_d$, $x < y$) to the abstraction. However, for larger counterexamples this might lead to an unnecessary blow-up of the state space, since refinement candidates can not only be atomic predicates, but also entire processes. Alternatively, a *single* candidate can be chosen in each refinement step. But this choice should be done with care. Imprudent decisions may guide the refinement in unfavourable directions; e.g., always selecting the *first* candidate when checking our safety property would first add the predicate $\neg empty_c$ and then iteratively all processes $Proc_2, \dots, Proc_m$ to the abstraction. The candidate $\neg empty_d$ would not be considered until it is validated that there is always a trace from location 1 to 2 in $Proc_1$. However, $\neg empty_d$ is the crucial predicate here: There is no process in our example system that ever sends a message to channel d. Hence, adding $\neg empty_d$ to our initial abstraction would already reveal that location 3 is not reachable, and we can deduce that **AG** $\neg(Proc_1@\texttt{Error})$ holds.

In this section we show how abstraction refinement, i.e. selecting the 'most promising' candidate, can be enhanced by heuristic guidance. In some cases it might be advisable to consider more than one unconfirmed counterexample; particularly when the generated counterexample only reveals a single candidate. For illustration consider again our example system and the liveness property **AF** $(Proc_1@\texttt{End})$. Model checking this formula on the aforementioned abstraction yields the unconfirmed counterexample shown below in (a). As we can see,

[2] In our approach, the initial abstraction is always given by the processes and predicates referenced in the temporal logic formula.

there is only one candidate ($empty_c$) and therefore no possibility for heuristic decisions. By disregarding the $\perp$-self-loop at location 1 for further unconfirmed counterexamples in the same abstraction, we obtain the additional counterexample shown in (b). Hence, we can enlarge the set of refinement candidates by generating several counterexamples for each abstraction level. In our approach, we perform this *multiple counterexample-generation* via excluding $\perp$-transitions that occurred in already discovered unconfirmed counterexamples.



Let us now consider an abstraction of a concurrent system, given by sets *Spot* and *Shade*, for which checking some temporal logic property yields *unknown*. Then by multiple counterexample-generation we can derive a set of refinement candidates $Candidates \subseteq Shade$. Now, the basis of our heuristic-guided refinement framework is the *abstraction dependence graph* (ADG):

Definition 8. *Let $Sys = \|_{i=1}^{n} Proc_i$ be a concurrent system with a spotlight abstraction given by the sets Spot and Shade. Moreover, let ψ be a temporal logic formula that evaluates to $\perp$ on this abstraction, and let $Candidates \subseteq Shade$ be a set of refinement candidates derived from unconfirmed counterexamples. Then the corresponding* abstraction dependence graph *is a tuple $ADG = (V, D)$ where*

- $V = Spot \cup \underbrace{Candidates \cup Shade(Proc)}_{\subseteq Shade}$ *is the set of vertices,*
  (note that V can also be repartitioned into subsets of Proc and Pred)
- $D \subseteq V \times V$ *is a dependence relation.*

Hence, the vertices of the abstraction dependence graph correspond to the elements of *Spot* and (a finite subset of) *Shade*. The ADG represents dependencies between the abstractions' components (processes as well as predicates), and thus can guide us in selecting promising refinement candidates. For computing the dependence relation D (and later our heuristic evaluation function) we introduce the following sets for each vertex $v \in V$:

- $Def(v)$: *set of shared variables/channels defined (modified) in v,*
  (note that $Def(v) = \varnothing$ if v corresponds to a predicate)
- $Ref(v)$: *set of shared variables/channels referenced in v,*
- $In(v, V')$ *: set of incoming edges into v from vertices in $V' \subseteq V$,*
- $Out(v, V')$ *: set of outgoing edges from v into vertices in $V' \subseteq V$.*

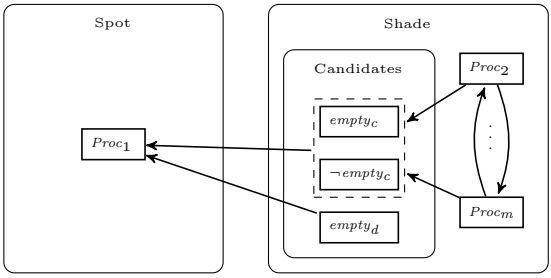
Now let $v, v' \in V, v \neq v'$ be a pair of vertices, then

$$\begin{aligned} D(v, v') \equiv \quad & v \in Proc \wedge \exists\, x\, (x \in Def(v) \wedge x \in Ref(v')) \\ \vee\ & v \in Pred \wedge v' \in Proc \wedge \exists\, x\, (x \in Ref(v) \wedge x \in Ref(v'))\,. \end{aligned}$$

If the relation $D(v, v')$ holds for a pair $v, v' \in V$ then we say: v' *depends on* v, or conversely: v *affects* v'.

Dependencies of particular interest are those between individual candidates and the spotlight. A candidate that affects a large number of spotlight components is likely a *beneficial* choice for refinement. Adding it to *Spot* would feed our abstraction with new details that are relevant for many spotlight processes and predicates. Thus, this choice might guide us closer to a definite result in verification. Contrary, a candidate with lots of dependencies within the shade might be a *costly* choice. By selecting such a candidate for refinement we would introduce several new dependencies between the spotlight and the shade, i.e. several new unknowns in our abstraction.

The abstraction dependence graph corresponding to our running example (with $Spot(Proc) = \{Proc_1\}$, $Spot(Pred) = \varnothing$, $\psi = \mathbf{AF}\ (Proc_1@\texttt{End})$ and the candidates derived from the two unconfirmed counterexamples (a) and (b)) is shown below:



Here, we have two candidates: $empty_c$[3] and $empty_d$. Now a very simple heuristic evaluation function $h : Candidates \to \mathbb{N}$ with respect to *benefits* and *costs* of possible refinement steps would be

$$h(v) \;=\; \underbrace{|Out(v, Spot)|}_{benefit(v)} - \underbrace{|In(v, Shade)|}_{cost(v)}$$

i.e. for a candidate v we compute the number of outgoing edges into the spotlight minus the number of incoming edges from the shade. The refinement procedure selects the candidate with the best evaluation value: $arg\ max_{v \in Candidates}\ h(v)$. Hence, in our example $empty_d$ is chosen due to fewer (no) dependencies within the shade. The subsequent verification run on the refined abstraction already reveals a definite result and refutes the liveness property $\mathbf{AF}\ (Proc_1@\texttt{End})$.

This example illustrates that even with simple heuristic evaluation functions we can guide the refinement in expedient directions, and thus obtain definite verification results on very small abstract models. Nevertheless, in the refinement framework used in our experiments we follow an enhanced approach to heuristic guidance. In particular, we construct abstraction dependence graphs extended with *weighted edges and vertices*. Edge weights allow us to comprise *quantitative aspects of dependence*; e.g. in terms of the number of variables/channels that are

[3] We can regard $\neg empty_c$ as a redundant candidate, since it is just a Boolean expression over $empty_c$. However, in our enhanced heuristics used in our experiments we also consider factors like the number of occurrences as a candidate for each predicate.

shared between pairs of processes. We moreover consider *transitive dependencies* in our heuristic decisions: Chains of dependencies within the shade contribute to the costs of a candidate, as well as direct dependencies. By vertex weights we express benefits and costs *apart from dependence*: A beneficial aspect of a refinement candidate is, e.g., the number of occurrences as a candidate in the generated set of counterexamples, whereas the size of a candidate (for processes: the number of its control flow locations) is a cost factor. A detailed description of the heuristics that we use in our experiments can be found in the next section.

5 Experimental Results

We have implemented our heuristic approach to abstraction refinement on top of our three-valued model checking tool 3Spot. In our experiments we compare four abstraction refinement heuristics. The first two heuristics are fairly naive: We generate one unconfirmed counterexample and then select the *first* encountered candidate (1Cex-1stCand), or alternatively, *all* candidates (1Cex-allCand) for refinement. In fact, 1Cex-1stCand is the 'heuristic' that we employed in our previous work [14]. In our advanced heuristics we use an evaluation function for selecting the presumably best candidate, once for a single counterexample (1Cex-bestCand) and once with multiple counterexample-generation (nCex-bestCand):

1Cex-1stCand	generate *one* unconfirmed counterexample select *first* candidate for refinement
1Cex-allCand	generate *one* unconfirmed counterexample select *all* candidates for refinement
1Cex-bestCand	generate *one* unconfirmed counterexample select *best* candidate v for refinement wrt. the evaluation function $h(v) = \underbrace{(occurrence(v) + linkingSpot(v))}_{benefit(v)} - \underbrace{(linkingShade(v) + size(v) + redundancy(v))}_{cost(v)}$
nCex-bestCand	generate N unconfirmed counterexamples (if existing) select *best* candidate for refinement wrt. the evaluation function h

As it can be seen, the evaluation function h that we use in our advanced heuristics is composed of five sub-functions:

- For a candidate v the function $occurrence(v)$ returns the number of occurrences as a candidate in the current set of unconfirmed counterexamples (note that a candidate can also occur several times in *one* counterexample).
- $linkingSpot(v)$ yields a value that characterises the *linking factor* of v with the spotlight; the linking factor incorporates the number of spotlight processes that are affected by the candidate, as well as the number of variables/channels that are shared between the candidate and each spotlight process:

  $$linkingSpot(v) = \sum_{v' \in Spot(Proc)} (D_w(v, v') + 0.5 \cdot D_w(v', v))$$

  Here D_w is a quantitative extension of the dependence relation D. $D_w(v, v')$ returns the *number* of variables/channels defined in v and referenced in v'.

– Accordingly, *linkingShade*(v) specifies the linking factor of v with the shade. In *linkingShade*(v) we incorporate the number, distance and size of shade processes that – possibly transitively, through other processes – affect the candidate; processes with a more indirect impact on the candidate (wrt. the vertex distance in the abstraction dependence graph) are weighted less:

$$linkingShade(v) = \sum_{v' \in Shade(Proc)} \frac{size(v')}{distance(v', v)}$$

Here $distance(v', v)$ returns the length of the shortest path from v' to v within the shade.
– The *size* of a candidate process corresponds to the number of predicates that are required to encode its control flow; the size of an atomic predicate is simply one.
– By the function *redundancy*(v) we counter a problem that frequently occurs when verifying concurrent systems with large-domain loop variables: The loop is completely unrolled by abstraction refinement, i.e. all possible predicates over the loop variable are added to the spotlight – though adding a certain process might already suffice to show termination of the loop. We solve this problem as follows: The more predicates over a distinct variable are in the spotlight, the higher we set the *redundancy value* (and hence the cost) of any candidate predicate over the same variable. Thus, a process affecting a loop variable eventually will have a better heuristic evaluation than new predicates over this variable. In this way, we can avoid the complete unwinding of loops for several verification tasks. *redundancy*(v) is defined for predicate candidates only:

$$redundancy(v) = |\{v' \in Spot(Pred) \mid Ref(v) \cap Ref(v') \neq \varnothing\}|$$

For our experiments, we have enriched the heuristic evaluation function with additional weights for every sub-function. Here, we put particular emphasis on *occurrence*(v) on the benefit-side and *linkingShade*(v) on the cost-side.

In our case study, we consider *multiple-resource allocation via message passing*. We look at systems with two classes of processes: *allocators* and *customers*. Each allocator manages the allocation of a *single* resource (though there may be more than one allocator per resource type), whereas every customer requests *several* resources before entering its critical section `Crit`. Moreover, allocators that provide a resource A can in turn be customers of a resource B, i.e. they have to acquire B first in order to provide A. This causes a high degree of transitive dependencies in our systems. Allocators and customers communicate via message passing, i.e. each resource type is associated with a distinct tuple of channels. The customers' individual resource demands and the orders of requests are randomly generated. Hence, our systems are *non-uniform* with respect to dependencies. The systems are parameterised in the number of resources, customers and allocators. We check the CTL liveness formula $\mathbf{AG}(\mathbf{AF}(Cust_i@\texttt{Crit}))$ for a randomly selected customer. A simple example of a multiple-resource allocation system with one customer and k allocators (resp. resources) is shown below:

$$requestR_1, releaseR_1, \ldots, requestR_k, releaseR_k : \texttt{channel [1] of int}$$

$$Cust_1 :: \left[\begin{array}{l}\texttt{loop forever}\\ \left[\begin{array}{l} send(requestR_1, 1)\\ \ldots\\ send(requestR_k, 1)\end{array}\right]\\ \texttt{Crit}\\ \left[\begin{array}{l} send(releaseR_1, 1)\\ \ldots\\ send(releaseR_k, 1)\end{array}\right]\end{array}\right] \quad \Big|\Big|_{i=1}^{k} Alloc_i :: \left[\begin{array}{l}\texttt{loop forever}\\ \left[\begin{array}{l} receive(requestR_i, x_i)\\ receive(releaseR_i, x_i)\end{array}\right]\end{array}\right]$$

Our experiments were performed on a 2.40 GHz Core 2 Duo Windows system with 3 GB memory. The results of our benchmark are shown in the table below. For small systems, where it is inevitable to take *all* processes and (predicates over) resources into the spotlight[4], the naive heuristics slightly outperform our advanced ones in verification time. The overhead when using 1Cex-bestCand and 2Cex-bestCand is caused by the additional computations (abstraction dependence graph, multiple counterexamples) for selecting the best refinement candidate. However, for larger systems (2 resources and more) this additional computations pay off: The verification under 2Cex-bestCand can be accomplished on significantly smaller abstractions. For the systems with 4 and 8 resources, we see that the advanced heuristics clearly outperform the naive ones in both size of the abstraction and verification time. With our advanced heuristic 2Cex-bestCand we are even successful and fast when verification under the other heuristics runs out of memory (OOM).

	System			Abstraction		
heuristic	resources	customers	allocators	$\|Spot(Proc)\|$	$\|Spot(Pred)\|$	time
1Cex-1stCand	1	2	1	3	1	0.74s
	2	5	3	6	1	6.94s
	3	6	4	6	1	8.83s
	4	8	8	–	–	OOM
	8	15	10	–	–	OOM
1Cex-allCand	1	2	1	3	1	0.67s
	2	5	3	5	2	7.06s
	3	6	4	5	4	195s
	4	8	8	5	4	307s
	8	15	10	–	–	OOM
1Cex-bestCand	1	2	1	3	1	1.39s
	2	5	3	5	2	16.4s
	3	6	4	6	2	27.3s
	4	8	8	5	4	83.9s
	8	15	10	–	–	OOM
2Cex-bestCand	1	2	1	3	1	1.67s
	2	5	3	3	2	6.47s
	3	6	4	3	2	6.52s
	4	8	8	3	2	8.55s
	8	15	10	4	3	57.5s

As we can see, our advanced refinement heuristic 2Cex-bestCand enriches our spotlight abstraction framework by great savings in both size of the final abstraction and verification time. We achieved similarly good results for checking safety ($\mathbf{AG}\neg(Cust_i@\texttt{Crit} \wedge Cust_j@\texttt{Crit})$) of multiple-resource allocation systems, and also for verifying concurrent systems with shared variables instead of channels.

[4] In small systems usually *all* processes are relevant for the checked property, and thus, a definite result is not possible without *all* these processes being in the spotlight.

6 Conclusion

In this paper we have introduced a heuristic-guided abstraction refinement framework for verifying concurrent systems. Though spotlight abstraction generally enables the verification of concurrent systems on very small abstract models, the detection of an adequate spotlight is a non-trivial task. The positive effect of abstraction is often compromised by bad refinement decisions: The refinement may be guided in unfavourable directions or the spotlight may be unnecessarily enlarged by redundant components. We have observed these drawbacks in our experiments under the *naive* refinement approach [14]. However, if there exists a small subset of system components which is sufficiently large for a definite result in verification, then this set will most likely be discovered by our novel approach to fully automatic abstraction refinement under *advanced heuristic guidance*. The only prize to pay is to generate multiple counterexamples and to perform additional dependency computations, which are the basis for heuristically selecting the *best* refinement candidate in every iteration. Finally, this allows us to exploit the full potential of spotlight abstraction for many verification tasks. We believe that our heuristic approach is universally helpful for verifying concurrent systems via spotlight abstraction. Nevertheless, as future work we intend to develop particular heuristics for different types of temporal logic properties, in order to further improve the performance of verification. Moreover, we plan to extend our approach to *parameterised verification of classwise symmetric systems* [17].

Related Work. The idea of using heuristics in model checking has received a lot of attention in research. In *directed model checking* (e.g. [6,12]) heuristics are employed to guide the exploration of the state space in order to obtain *counterexamples of minimal length*. In contrast, our approach aims at minimising the size of the abstraction on which the property of interest can be proven. Abstraction refinement under heuristic guidance has been considered in [10] and [9] for verifying *hardware designs*. Their framework is not based on predicate abstraction but on *variable hiding*. Moreover, their heuristics aim at finding a minimal solution for the state separation problem [10], resp. maximising the number of spurious counterexamples that can be ruled out in one refinement step [9]. Using heuristics for the verification of *concurrent systems* has also been considered in [16]. Therein, heuristics for automatically selecting constraints that rule out infeasible interleavings in a FLAVERS [16] model are presented. The heuristics are based on the system, the property, and other constraints that have to be selected manually. Hence, their approach is neither fully automatic nor integrated into an iterative, counterexample-based refinement framework. Another work related to ours is that of Fecher and Shoham [7] who define heuristics for local refinement in the context of *lazy abstraction*: New predicates are only locally added, i.e. at single abstract states. Their approach is not tailored to concurrent systems and they do not report experimental results which hampers a comparison with our method. However, the combination of heuristic-guided spotlight abstraction and lazy abstraction is another interesting direction for future research.

Finally, our work is related to other state space reduction techniques for concurrent systems. Approaches based on *symmetry reduction* [5,13] exploit the

uniform structure of parameterised systems in order to obtain smaller abstractions. These reductions are not applicable to *non-uniform* systems (composed of heterogeneous processes). Our approach focuses on finite-state concurrent systems but does not restrict the systems to be uniform. In fact, we exploit information about potentially *non-uniform* system structures for our heuristic decisions.

References

1. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of c programs. SIGPLAN Not. 36(5), 203–213 (2001)
2. Ball, T., Podelski, A., Rajamani, S.K.: Relative Completeness of Abstraction Refinement for Software Model Checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 158–172. Springer, Heidelberg (2002)
3. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM TPLS 8, 244–263 (1986)
4. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-Guided Abstraction Refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
5. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
6. 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)
7. Fecher, H., Shoham, S.: Local abstraction-refinement for the mu-calculus. STTT 13(4), 289–306 (2011)
8. Fitting, M.: Kleene's three valued logics and their children. Fundamenta Informaticae 20(1-3), 113–131 (1994)
9. Glusman, M., Kamhi, G., Mador-Haim, S., Fraer, R., Vardi, M.Y.: Multiple-Counterexample Guided Iterative Abstraction Refinement: An Industrial Evaluation. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 176–191. Springer, Heidelberg (2003)
10. He, F., Song, X., Gu, M., Sun, J.: Heuristic-guided abstraction refinement. Comput. J. 52(3), 280–287 (2009)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL 2004, pp. 232–244. ACM, New York (2004)
12. Hoffmann, J., Smaus, J.-G., Rybalchenko, A., Kupferschmid, S., Podelski, A.: Using Predicate Abstraction to Generate Heuristic Functions in UPPAAL. In: Edelkamp, S., Lomuscio, A. (eds.) MoChart IV. LNCS (LNAI), vol. 4428, pp. 51–66. Springer, Heidelberg (2007)
13. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
14. Schrieb, J., Wehrheim, H., Wonisch, D.: Three-Valued Spotlight Abstractions. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 106–122. Springer, Heidelberg (2009)
15. Shoham, S., Grumberg, O.: 3-valued abstraction: More precision at less cost. Inf. Comput. 206, 1313–1333 (2008)
16. Tan, J., Avrunin, G., Clarke, L.: Heuristic-based model refinement for flavers. In: ICSE 2004, pp. 635–644 (2004)
17. Timm, N., Wehrheim, H.: On Symmetries and Spotlights – Verifying Parameterised Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 534–548. Springer, Heidelberg (2010)
18. Wachter, B., Westphal, B.: The Spotlight Principle. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 182–198. Springer, Heidelberg (2007)

More Anti-chain Based Refinement Checking*

Ting Wang[1], Songzheng Song[2], Jun Sun[3], Yang Liu[2], Jin Song Dong[2], Xinyu Wang[1,**], and Shanping Li[1]

[1] College of Computer Science and Technology, Zhejiang University
{qdw,wangxinyu,shan}@zju.edu.cn
[2] National University of Singapore
{songsongzheng@,tslliuya@,dongjs@comp.}nus.edu.sg
[3] Singapore University of Technology and Design
sunjun@sutd.edu.sg

Abstract. Refinement checking plays an important role in system verification. It establishes properties of an implementation by showing a refinement relationship between the implementation and a specification. Recently, it has been shown that anti-chain based approaches increase the efficiency of trace refinement checking significantly. In this work, we study the problem of adopting anti-chain for stable failures refinement checking, failures-divergence refinement checking and probabilistic refine checking (i.e., a probabilistic implementation against a non-probabilistic specification). We show that the first two problems can be significantly improved, because the state space of the product model may be reduced dramatically. Though applying anti-chain for probabilistic refinement checking is more complicated, we manage to show improvements in some cases. We have integrated these techniques into the PAT model checking framework. Experiments are conducted to demonstrate the efficiency of our approach.

1 Introduction

Model checking has established itself as an effective technique for system verification. It works by exhaustively searching through the state space in order to show that an implementation model, in certain modeling language, satisfies a property. Properties are often specified using temporal logic formulae such as CTL or LTL, in other words, a language different from the modeling language. An alternative approach is called refinement checking. Different from temporal-logic based model checking, refinement checking shows a refinement relationship between two models in the same language, one modeling an implementation and one modeling a specification. If the specification satisfies certain property and the refinement relationship is strong enough to preserve the property, we imply that the property is satisfied by the implementation. A variety of refinement relationships have been defined, which preserve different classes of properties. For instance, safety can be verified by showing a trace refinement relationship. Combination of safety and liveness is verified by showing a stable failures

* This research is sponsored in part by NSFC Program (No.61103032) and 973 Program (No.2009CB320701) of China.

** Corresponding author.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 364–380, 2012.

refinement relationship if the system is divergence-free or otherwise by showing a failures-divergence refinement relationship. The readers are refer to [12] for a discussion on the expressiveness of different refinement.

Refinement checking has been traditionally used to verify CSP [11]. The success of the FDR refinement checker [1], which supports fully automatic checking of the above-mentioned refinement relationships, evidences the usefulness of refinement checking. Recently, *Sun et al.* extended the idea of automated trace refinement checking to probabilistic systems [14], which we refer to as probabilistic refinement checking in this work[1]. The idea is that, given a probabilistic implementation model (which has the semantics of a Markov Decision Process) and a non-probabilistic specification model, probabilistic refinement checking calculates the probability of the implementation exhibiting traces of the specification model. This is useful as, for instance, if the specification model captures desired system behaviors, the result is the probability of the implementation behaving 'well'.

Due to the non-determinism in the specification, refinement checking often relies on the classic subset construction approach. The subset construction is used to build a deterministic finite-state automaton (DFA) from the specification, which is in general a non-deterministic finite-state automaton (NFA). Next, refinement checking works by computing the synchronous product of the implementation and transforming the problem into a reachability analysis problem in the product. In the worst case, the resultant DFA could have exponentially more states than the original NFA. As a result, refinement checking suffers from state space explosion. Recently, Wulf *et al.* proposed an approach (for solving the language universality problem and trace refinement checking) named anti-chain [16]. It has been shown that this approach outperforms the previous ones significantly. The key point of anti-chain based approaches is that the complete subset construction and computing the complete state space of the product are avoided. Given that the existing approaches for checking other refinement relationships are all based on the subset construction, it is only naturally to investigate whether anti-chain can be used for better performance as it did for trace refinement checking.

In this work, we study three kinds of refinement checking, in particular, stable failures refinement, failures-divergence refinement and probabilistic refinement checking. The problem is non-trivial as we need to formally prove that anti-chain works with stable failures semantics and failures-divergence semantics. Furthermore, it is complicated for probabilistic refinement checking as omitting parts of the product would affect the probability. We make the following technical contribution. Firstly, we show that anti-chain can be readily used to improve stable failures refinement and failures-divergence refinement. Secondly, we show that anti-chain can be used to improve probabilistic refinement checking in some particular cases, using an iterative probability calculation method. Lastly, we implement the technique in the PAT model checker [13] and show improvement over existing approaches (significant for stable failures refinement and failures-divergence refinement).

Related Works. This work is related to research on anti-chain based model checking. Wulf *et al.* proposed the anti-chain based approach for checking the language universality and trace refinement of NFA [16]. It has been shown that the anti-chain based

[1] Probabilistic refinement has been used by different researchers to mean different things.

approach may outperform the standard ones by several orders of magnitude. Their following works show that significant improvements can also be brought to the model checking problem of LTL by using anti-chain based algorithms [8, 17]. Later Abdulla *et al.* improved the approach through exploiting a simulation relation on the states of NFA [2]. Remotely related are anti-chain based methods for solving other problems, e.g., the LTL realizability and synthesis problem [7, 10] and the universality and language inclusion problem of tree automata [2, 6]. In our work, we focus on stable failures refinement, failures-divergence refinement and probabilistic refinement checking.

Organization. Section 2 reviews trace refinement and anti-chain based trace refinement checking. Section 3 presents algorithms for anti-chain based stable failures refinement checking and failures-divergence refinement checking. Section 4 shows that anti-chain can be used to improve probabilistic refinement checking (with a non-probabilistic specification model). Lastly, Section 5 concludes the paper.

2 Background

In this section, we review previous work on anti-chain based trace refinement checking.

2.1 Trace Refinement

Let Σ be a set of event names; τ denote an invisible event; and Σ_τ denote $\Sigma \cup \{\tau\}$.

Definition 1 (LTS). *A labeled transition system (LTS) is a tuple $\mathcal{L} = (S, init, Act, T)$ where S is a set of states; $init \in S$ is an initial state; $Act \subseteq \Sigma_\tau$ is a set of events and $T : S \times Act \times S$ is a labeled transition relation.*

For simplicity, $(s, e, s') \in T$ is sometimes written as $s \xrightarrow{e} s'$. An LTS is deterministic if and only if for all $s \in S$ and $s \in \Sigma_\tau$, if $s \xrightarrow{e} u$ and $s \xrightarrow{e} v$, then $u = v$. We write $enable(s)$ to denote the set $\{e \mid \exists\, s'.\ s \xrightarrow{e} s'\}$. We write $u \rightsquigarrow v$ if there exists a finite sequence of states $\langle s_0, s_1, \cdots, s_n\rangle$ such that $s_i \xrightarrow{\tau} s_{i+1}$ for all i and $u = s_0$ and $v = s_n$. We write $u \overset{e}{\rightsquigarrow} v$ if $u \rightsquigarrow u'$ and $u' \xrightarrow{e} v'$ and $v' \rightsquigarrow v$. A finite sequence of events $\langle e_0, e_1, \cdots, e_n\rangle$ is a trace of $\mathcal{L}$ if and only if there exists a sequence of state $\langle s_0, s_1, \cdots, s_n\rangle$ such that $s_i \overset{e_i}{\rightsquigarrow} s_{i+1}$ for all i and $s_0 = init$. The traces of $\mathcal{L}$ are denoted as $traces(\mathcal{L})$.

Definition 2 (LTS Synchronous Product). *Let $\mathcal{L}_i = (S_i, init_i, Act_i, T_i)$ where $i \in \{1, 2\}$ be two LTSs such that $\tau \notin Act_2$. The synchronous product of $\mathcal{L}_1$ and $\mathcal{L}_2$, written as $\mathcal{L}_1 \times \mathcal{L}_2$, is an LTS $\mathcal{L} = (S, init, Act, T)$ such that $S = S_1 \times S_2$; $init = (init_1, init_2)$; $Act = Act_1 \cup Act_2$; and T is the minimum labeled transition relation satisfying the following conditions.*

- *If $(s_1, \tau, s_1') \in T_1$, $((s_1, s_2), \tau, (s_1', s_2)) \in T$ for all $s_2 \in S_2$;*
- *If $(s_1, e, s_1') \in T_1$ and $(s_2, e, s_2') \in T_2$ and $e \notin \tau$, $((s_1, s_2), e, (s_1', s_2')) \in T$.*

Notice that all events except τ are to be synchronized by the two LTSs.

Definition 3 (Trace Refinement). *Let $\mathcal{L}_i$ where $i \in \{1,2\}$ be two LTSs. $\mathcal{L}_1$ trace-refines $\mathcal{L}_2$ if and only if $traces(\mathcal{L}_1) \subseteq traces(\mathcal{L}_2)$.*

The standard approach for trace refinement is based on the subset construction. That is, the specification LTS_2 is transformed into trace-equivalent deterministic LTS without τ-transitions through the process of determinization. Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. The determinized LTS of $\mathcal{L}$ is $det(\mathcal{L}) = (S', init', Act', T')$ where $S' \subseteq 2^S$ is a set of sets of states, $init' = \{s \mid init \rightsquigarrow s\}$; $Act' = Act \setminus \{\tau\}$ and T' is a transition relation satisfying the following condition: $(N, e, N') \in T'$ if and only if $N' = \{s' \mid \exists\, s : N.\ s \overset{e}{\rightsquigarrow} s'\}$. Notice that states which can be reached via the same trace are grouped together in $det(\mathcal{L})$.

Given an implementation $\mathcal{L}_1$ and a specification $\mathcal{L}_2$, the standard trace refinement checking is to construct (often on-the-fly) the product $\mathcal{L}_1 \times det(\mathcal{L}_2)$ and then try to construct a state of the product (s_1, s_2) (where s_1 is a state of $\mathcal{L}_1$ and s_2 is a set of states in $\mathcal{L}_2$) such that s_2 is an empty set. Such a 'co-witness' state is called a TR-witness state. In the worst case, this algorithm has a complexity exponential in the number of states of $\mathcal{L}_2$.

2.2 Trace Refinement Checking with Anti-chain

It has been shown that trace refinement checking based on anti-chain offers significantly better performance [16]. Given two LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, the anti-chain method explores a 'simulation' relation in $\mathcal{L}_1 \times det(\mathcal{L}_2)$. Given any two states (s_1, s_2) and (s_1', s_2') of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, let $(s_1', s_2') \leq (s_1, s_2)$ denote $s_1 = s_1'$ and $s_2 \subseteq s_2'$.

Proposition 1. *If $(s_1', s_2') \leq (s_1, s_2)$ and $(s_1, s_2) \overset{e}{\rightarrow} (u, v)$, then there exists (u', v') such that $(s_1', s_2') \overset{e}{\rightarrow} (u', v')$ and $u' = u$ and $v \subseteq v'$.* □

By the above proposition, it can be readily shown that a TR-witness state is reachable from (s_1', s_2') implies that a TR-witness state must be reachable from (s_1, s_2). As a result, if (s_1, s_2) has been explored, we can skip (s_1', s_2').

Formally, an anti-chain is a set A of sets such that $x \not\subseteq y$ and $y \not\subseteq x$ for all $x \in A$ and $y \in A$, i.e., any pair of sets in A are incomparable. An anti-chain supports two operations. One is to check whether it contains a subset of a given set. let x be the given set, we denote $x \Subset A$ if and only if there exists $y \in A$ such that $y \subseteq x$. The other is to add a given set x in A. $A \Cup x$ is defined as $\{y \mid y \in A \wedge x \not\subseteq y\} \cup \{x\}$, i.e., $A \Cup x$ contains x and all sets in A which is not a superset of x. Obviously, an empty set is an anti-chain by definition.

Algorithm 1 shows the anti-chain based algorithm. In an abuse of notation, we write $(s, X) \Subset A$ to denote that the set $(\{s\} \cup X) \Subset A$; and $A \Cup (s, X)$ to denote $A \Cup (\{s\} \cup X)$. The algorithm works as follows. After initialization, the algorithm pops one state $(impl, spec)$ from $working$ and adds it to the set $antichain$, and then generates all successors of the state and adds them to $working$ unless $(impl', spec') \Subset antichain$ is true, till the stack $working$ is empty or a TR-witness state is found. We remark that $antichain$ keeps to be an anti-chain during this algorithm, because line 5 and line 13 guarantee there are no subsets or supersets of the new added state in the updated $antichain$. Soundness of the algorithm can be referred to in [2] [16].

Algorithm 1 Trace Refinement Checking Algorithm with Anti-chain

1: let $working$ be a stack containing a pair $(init_1, \{s \mid init_2 \rightsquigarrow s\})$;
2: let $antichain := \varnothing$;
3: **while** $working \neq \varnothing$ **do**
4: pop $(impl, spec)$ from $working$;
5: $antichain := antichain \Cup (impl, spec)$;
6: **for all** $(impl, e, impl') \in T_1$ **do**
7: **if** $e = \tau$ **then**
8: $spec' := spec$;
9: **else**
10: $spec' := \{s' \mid \exists\, s \in spec.\ s \overset{e}{\rightsquigarrow} s'\}$;
11: **if** $spec' = \varnothing$ **then**
12: **return** false;
13: **if** $(impl', spec') \Subset antichain$ is not true **then**
14: push $(impl', spec')$ into $working$;
15: **return** true;

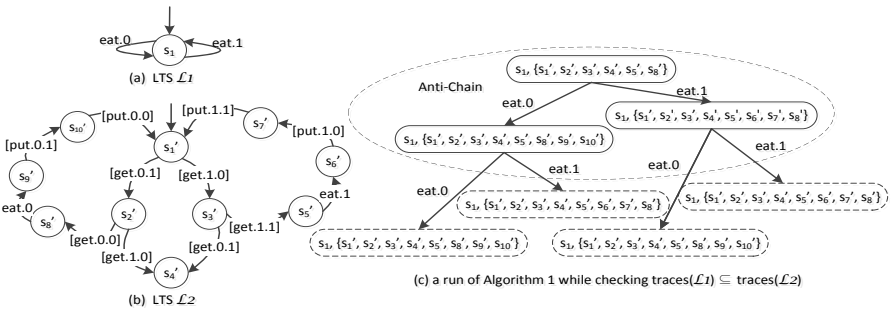


Fig. 1. Trace Refinement Checking Algorithm with Anti-Chain

Theorem 1. *[16] Algorithm 1 returns true if and only if* $traces(\mathcal{L}_1) \subseteq traces(\mathcal{L}_2)$. □

Example 1. Figure 1 shows a simple example of dining philosopher [11] to demonstrate how Algorithm 1 works. The problem is summarized as N philosophers sitting around a round table with a single fork between each pair, and each philosopher requiring both neighboring forks to eat. The LTS $\mathcal{L}_2$ in Figure 1 (b) shows the complete state graph of the system with two philosophers. The invisible events is denoted as $[event]$, i.e., $[get.0.1]$ means that the hidden event is philosopher 0 getting the right fork which is represented as 1. Note that checking $traces(\mathcal{L}_1) \subseteq traces(\mathcal{L}_2)$ is to confirm whether every philosopher can eat. From Figure 1 (c) we can see that the search does not continue from the state $(s_1, \{s_1', s_2', s_3', s_4', s_5', s_8', s_9', s_{10}'\})$ because $\{s_1', s_2', s_3', s_4', s_5', s_8'\} \subseteq \{s_1', s_2', s_3', s_4', s_5', s_8', s_9', s_{10}'\}$. In this case, Algorithm 1 generates 3 states which are labeled with $Anti{-}Chain$, while the classical algorithm based on subset construction generates 7 states. □

Algorithm 2 Stable Failures Refinement Checking Algorithm with Anti-chain

```
1: let working be a stack containing a pair (init_1, {s | init_2 ⇝ s});
2: let antichain := ∅;
3: while working ≠ ∅ do
4:     pop (impl, spec) from working;
5:     antichain := antichain ⩏ (impl, spec);
6:     if refusals(impl) ⊈ refusals(spec) then
7:         return false;
8:     for all (impl, e, impl') ∈ T_1 do
9:         if e = τ then
10:            spec' := spec;
11:        else
12:            spec' := {s' | ∃ s ∈ spec. s ⇝^e s'};
13:        if spec' = ∅ then
14:            return false;
15:        if (impl', spec') ∈ antichain is not true then
16:            push (impl', spec') into working;
17: return true;
```

3 Failures/Divergence Refinement Checking with Anti-chain

In this section, we demonstrate that anti-chain can be used to improve stable failures refinement checking and failures-divergence refinement checking.

3.1 Stable Failures Refinement Checking

Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. Given a state $s \in S$, s is stable if $\tau \notin enable(s)$. Given a stable state s, the refusals of s, written as $refusals(s)$, is defined as $\{X \mid \exists\, s'.\ s \rightsquigarrow s' \wedge \tau \notin enable(s') \wedge X \subseteq \Sigma \setminus enable(s')\}$. The failures of $\mathcal{L}$, written as $failures(\mathcal{L})$, is defined as $\{(tr, X) : \Sigma^* \times 2^{\Sigma} \mid \exists\, s.\ init \overset{tr}{\rightsquigarrow} s \wedge X \in refusals(s)\}$ where $init \overset{tr}{\rightsquigarrow} s$ denotes that there exists a run $\langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1}\rangle$ such that $s_0 = init$ and $s_{n+1} = s$ and $tr = \langle e_0, e_1, \cdots, e_n\rangle$.

Definition 4 (Stable Failures Refinement). *Let $\mathcal{L}_i$ where $i \in \{1, 2\}$ be two LTSs. $\mathcal{L}_1$ refines $\mathcal{L}_2$ in stable failures semantics if and only if $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.*

The existing stable failures refinement checking algorithm [1] works by searching for a state (x, y) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$ such that $y = \varnothing$ or $refusals(x) \not\subseteq refusals(y)$. Such a state is called a SFR-witness state. In the following, we extend Algorithm 1 for stable failures refinement checking. Given a set states x, we write $refusals(x)$ to denote $\{r \mid \exists\, s \in x.\ r \in refusals(s)\}$. The algorithm is shown in Algorithm 2.

Lemma 1. *For every state (s_1, s_2) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, for all (s_1, s_2') in the product, if $s_2' \subseteq s_2$, then a SFR-witness state is reachable from (s_1, s_2) implies a SFR-witness state is reachable from (s_1, s_2').*

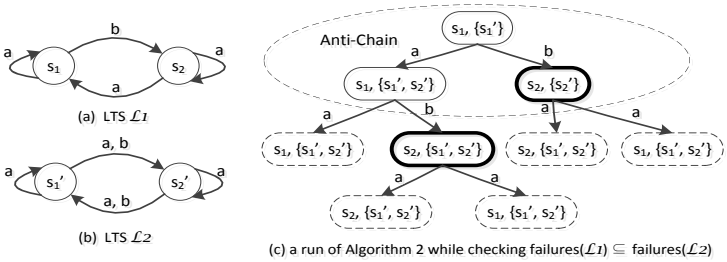


Fig. 2. Stable Failures Refinement Checking Algorithm with Anti-Chain

Proof: By induction. The base case is that (s_1, s_2) is a SFR-witness state, then $s_2 = \varnothing$ or $refusals(s_1) \not\subseteq refusals(s_2)$. Because $s_2' \subseteq s_2$ by assumption, $refusals(s_2') \subseteq refusals(s_2)$. Then we have $s_2' = \varnothing$, or $refusals(s_1) \not\subseteq refusals(s_2')$. Thus, (s_1, s_2') is a SFR-witness state. Next, we prove the induction step. Assume that (s_1, s_2) satisfies the condition, i.e., for any state (s_1, s_2') such that $s_2' \subseteq s_2$, a SFR-witness state is reachable from (s_1, s_2) implies a SFR-witness state is reachable from (s_1, s_2'). Let (x, y) be a state of the product such that $(x, y) \xrightarrow{e} (s_1, s_2)$. For all (x, y') such that $y' \subseteq y$, we can get $(x, y') \xrightarrow{e} (s_1, s_2')$ such that $s_2' \subseteq s_2$. Therefore, the induction step holds by induction hypothesis. Thus, the lemma is true. □

Theorem 2. *Algorithm 2 returns true if and only if* $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.

Proof For a state S of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, define $Dist(S) \in N \cup \{\infty\}$ as the length of the shortest SFR-witness trace from S (if a SFR-witness state is not reachable from S, $Dist(S) = \infty$). For a set of states $States$, if $States = \varnothing$, $Dist(States) = \infty$, otherwise, $Dist(States) = min_{S \in States}\, Dist(S)$. The predicate $SFR(States)$ is true if and only if all the states in $States$ are not SFR-witness states. Then the correctness of Algorithm 2 can be proved using the two invariants below. The invariants can be proved in a very similar way to [2].

1. $\neg SFR(antichain \cup working) \Rightarrow \neg SFR(\{(i, \{s \mid init_2 \rightsquigarrow s\}) \mid i \in init_1\})$.
2. $\neg SFR(\{(i, \{s \mid init_2 \rightsquigarrow s\}) \mid i \in init_1\}) \Rightarrow Dist(antichain) > Dist(working)$.

Because the number of state is finite and all states are only visited once, Algorithm 2 eventually terminates. Algorithm 2 returns false only if the state $spec'$ is an empty set on line 13, or $(impl, spec)$ satisfies the condition $refusals(impl) \not\subseteq refusals(spec)$ on line 6. The former case has been proved in Algorithm 1. In the latter case, $(impl, spec)$ is a SFR-witness state, and hence $SFR(antichain \cup working)$ is false. By invariant 1, $\mathcal{L}_1$ cannot refine $\mathcal{L}_2$ in stable failures semantics. Algorithm 2 returns true only when $working$ is empty, which implies that $Dist(antichain) > Dist(working)$ is not true. By invariant 2, $\mathcal{L}_1$ refines $\mathcal{L}_2$ in stable failures semantics. □

Example 2. If Algorithm 2 is applied to the example presented in Figure 1, the reduction remains the same as for trace refinement checking (from 7 states to 3 states). We show another example with some counterexamples, as shown in Figure 2. Notice

that the refusal set of s_2 is $\{b\}$, and s_1, s_1', s_2' do not refuse any event. Then the two states with bold circles are SFR-witness states. Since $\{s_1'\} \subseteq \{s_1', s_2'\}$, the search does not continue from the state $(s_1, \{s_1', s_2'\})$. We can see that a SFR-witness state which is reachable from $(s_1, \{s_1', s_2'\})$ is also reachable from $(s_1, \{s_1'\})$ after the pruning of states. In this case, Algorithm 2 generates 3 states which are labelled with $Anti{-}Chain$, while the classical algorithm may generate 4 states before finding a SFR-witness state. Moreover, Algorithm 2 may find a shorter witness trace than the classical algorithm. □

3.2 Failures-Divergence Refinement Checking

In the following, we show how to adopt anti-chain for failures-divergence refinement checking. Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. Given a state s, s diverges if and only if s can performance an infinite number of τ-transitions. A trace tr is divergent, written as $div(tr)$, if and only if there exists a prefix pre of tr or tr itself such that $init \overset{pre}{\rightsquigarrow} s$ and s diverges. We write $divergences(\mathcal{L})$ to be $\{tr \mid div(tr)\}$.

Definition 5 (Failures-Divergence Refinement). *Let $\mathcal{L}_i$ where $i \in \{1, 2\}$ be two LTSs. $\mathcal{L}_1$ refines $\mathcal{L}_2$ in failures-divergence semantics if and only if $divergences(\mathcal{L}_1) \subseteq divergences(\mathcal{L}_2)$ and $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.*

In the following, we extend Algorithm 2 for failures-divergence refinement checking. The algorithm is shown in Algorithm 3. Given a set of state x, we say that x diverges if there exists $s \in x$ such that s diverges. Like in the existing failures-divergence refinement checking algorithm [1], the idea is to search for a FDR-witness state (x, y) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$ such that $y = \varnothing$ or $refusals(x) \not\subseteq refusals(y)$ or x diverges but not y.

Lemma 2. *For every state (s_1, s_2) of $\mathcal{L}_1 \times det(\mathcal{L}_2)$, for all (s_1, s_2') in the product, if $s_2' \subseteq s_2$, then a FDR-witness state is reachable from (s_1, s_2) implies a FDR-witness state is reachable from (s_1, s_2').*

Proof: By induction. The base case is that (s_1, s_2) is a FDR-witness state, then $s_2 = \varnothing$ or $refusals(s_1) \not\subseteq refusals(s_2)$ or s_1 diverges and s_2 does not. Because $s_2' \subseteq s_2$ by assumption, $refusals(s_2') \subseteq refusals(s_2)$ and if s_2' diverges, so does s_2. Thus, (s_1, s_2') is a SFR-witness state since $s_2 = \varnothing$ implies $s_2' = \varnothing$; $refusals(s_1) \not\subseteq refusals(s_2)$ implies $refusals(s_1) \not\subseteq refusals(s_2')$; and s_2 not divergent implies that s_2' does not diverge either. Next, we prove the induction step. Assume that (s_1, s_2) satisfies the condition. Let (x, y) be a state of the product such that $(x, y) \xrightarrow{e} (s_1, s_2)$. For all (x, y') such that $y' \subseteq y$, we can get $(x, y') \xrightarrow{e} (s_1, s_2')$ such that $s_2' \subseteq s_2$. Therefore, the induction step holds by induction hypothesis. Thus, the lemma is true. □

Theorem 3. *Algorithm 3 returns true if and only if $divergences(\mathcal{L}_1) \subseteq divergences(\mathcal{L}_2)$ and $failures(\mathcal{L}_1) \subseteq failures(\mathcal{L}_2)$.*

Proof Define $Dist(S) \in N \cup \{\infty\}$ as the length of the shortest FDR-witness trace from a state S of $\mathcal{L}_1 \times det(\mathcal{L}_2)$(if a FDR-witness state is not reachable from S, $Dist(S) = \infty$). Given a set of states $States$, if $States = \varnothing$, $Dist(States) = \infty$, otherwise, $Dist(States) = min_{S \in States}\, Dist(S)$. The predicate $FDR(States)$ is true if and only if all the states in $States$ are not FDR-witness states. The correctness of Algorithm 3 can be proved similarly as for Algorithm 2, using the following two invariants.

Algorithm 3 Failures-Divergence Refinement Checking Algorithm with Anti-chain

```
1: let working be a stack containing a pair (init_1, {s | init_2 ⇝ s});
2: let antichain := ∅;
3: while working ≠ ∅ do
4:     pop (impl, spec) from working;
5:     antichain := antichain ⩏ (impl, spec);
6:     if impl diverges then
7:         if spec does not diverge then
8:             return false;
9:     else
10:        if refusals(impl) ⋦ refusals(spec) then
11:            return false;
12:        for all (impl, e, impl') ∈ T_1 do
13:            if e = τ then
14:                spec' := spec;
15:            else
16:                spec' := {s' | ∃ s ∈ spec. s -e-> s'};
17:            if spec' = ∅ then
18:                return false;
19:            if (impl', spec') ∉ antichain is not true then
20:                push (impl', spec') into working;
21: return true;
```

1. $\neg FDR(antichain \cup working) \Rightarrow \neg FDR(\{(i, \{s \mid init_2 \rightsquigarrow s\}) \mid i \in init_1\})$.
2. $\neg FDR(\{(i, \{s \mid init_2 \rightsquigarrow s\}) \mid i \in init_1\}) \Rightarrow Dist(antichain) > Dist(working)$.

□

Example 3. We use the example shown in Figure 3 to demonstrate how algorithm 3 works. The state s_2 in LTS $\mathcal{L}_1$ has a self-loop labeled with τ. The two states with bold circles are FDR-witness states now. Since $\{s_1'\} \subseteq \{s_1', s_2'\}$, the search does not continue from the state $(s_1, \{s_1', s_2'\})$. We can see that a FDR-witness state which is reachable from $(s_1, \{s_1', s_2'\})$ is also reachable from $(s_1, \{s_1'\})$ after pruning the states. □

3.3 Implementation and Evaluation

The proposed algorithms have been adopted in the Process Analysis Toolkit (PAT) [13]. PAT is designed for systematic validation of distributed/concurrent systems using state-of-the-art model checking techniques. In the following, we evaluate the performance of the algorithms using a range of real-life parameterized systems. All the systems are embedded in the PAT package and available online. The data is obtained with Intel(R) Core(TM) i7-2640M CPU at 2.80GHz and 8GB RAM.

The pairs of LTSs (one as implementation and one as specification) are generated from different systems or same systems with different parameters. The systems include a multi-valued register simulation system with one reader or multiple readers [4], an implementation of concurrent stack with or without linearization point [15], a mail-box system [3], a system of scalable nonzero indicator [9] and the dining philosopher

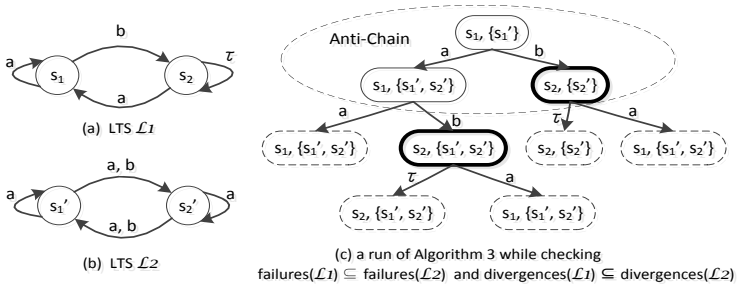


Fig. 3. Failures-Divergence Refinement Checking Algorithm with Anti-Chain

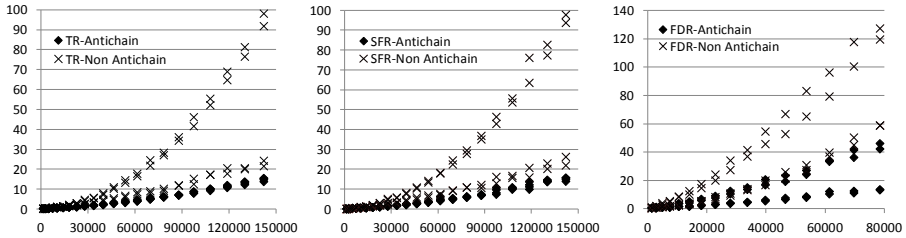


Fig. 4. Refinement Checking Results of Concurrent Stack Implementation

problem [11]. In total about 300 pairs of LTS were generated to compare the anti-chain algorithms and the classical ones for all three kinds of refinement checking.

Figure 4 shows the statistics of a typical example, i.e., the concurrent stack, from which we can see significant performance improvement. In the figure, the horizontal axis is the sum of the sizes of the two LTSs for refinement checking, and the vertical axis is the execution time (in seconds) of the corresponding algorithm. Each point shows the checking time for a pair of LTSs. We can see that for all three kinds of refinement checking, anti-chain based algorithms offer significantly better performance. The complete experimental results, with the refinement checking assertions always being valid, are summarized in Table 1. Notice that if the pair of LTSs are equivalent in terms of traces or failures or failures-divergence, we can perform the refinement checking in both directions. This is shown in the table using two columns $\subseteq$ and $\supseteq$. A few cases are marked as '—' as the result is false, which are discussed later. Furthermore, '$unknown$' means either out of memory or running for more than 30 minutes. It can be observed that the speedup differs for different systems. In most cases, the anti-chain approach has a much better performance than the classical one, e.g., in the concurrent stack linearization point implementation, it is 30.28 times faster for stable failures refinement checking and 12.16 times faster for failures-divergence refinement checking. Moreover, the larger the system is, the larger the speedup is. In some cases, anti-chain can not reduce the number of states at all simply because the specifications are deterministic. In some cases (e.g., SNZI), although anti-chain reduces the number of states, the benefit is not significant enough to overcome the computational overhead of the anti-chain operations defined in section 2.2.

Table 1. Testing on Refinement Checking Assertions which are valid

System	Size	Trace (Speedup)		Stable Failures (Speedup)		Failures-Divergence (Speedup)	
		$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$	$\supseteq$
Multi-valued Register	0-10000	2.21	1.42	2.32	1.63	3.48	1.46
Simulation with	10000-100000	4.54	2.14	4.45	2.09	6.61	1.73
1 Reader and 1 Writer	100000-700000	6.74	2.88	6.64	2.92	unknown	2.59
Multi-valued Register	0-10000	2.17	1.49	1.99	1.45	3.33	1.43
Simulation with	10000-100000	3.32	2.05	3.32	2.11	3.55	1.70
Multiple Readers	100000-700000	6.45	2.68	6.14	2.72	unknown	3.16
Concurrent	0-10000	1.48	1.85	1.63	1.72	2.26	1.60
Stack	10000-100000	1.70	3.72	1.71	3.71	2.71	3.22
Implementation	100000-200000	1.62	5.90	1.56	6.44	2.85	4.96
Concurrent Stack	0-10000	0.75	5.33	—	5.99	—	2.70
Linearization Point	10000-30000	0.84	13.94	—	14.11	—	5.00
Implementation	30000-60000	0.91	30.37	—	30.28	—	12.16
Mailbox	0-700000	1.13	1.54	1.11	1.61	1.39	1.01
SNZI	0-50000	0.89	2.45	0.93	2.42	1.03	1.14
Dining Philosopher	0-50000	0.99	1.14	—	1.02	—	1.17

Table 2. Testing on Refinement Checking Assertions which are invalid

System	Size	Trace With AC(s)	Trace W/o AC(s)	Stable Failures With AC(s)	Stable Failures W/o AC(s)	Failures-Divergence With AC(s)	Failures-Divergence W/o AC(s)
Multi-valued	3175	0.10	0.42	0.09	0.43	0.44	2.88
Register	24655	1.52	12.12	1.71	12.02	9.78	146.95
Simulation with	117288	3.92	136.37	3.57	138.50	48.52	985.24
Multiple Readers	194455	22.11	294.71	21.51	299.21	180.91	unknown

In presence of counterexamples, anti-chain based algorithms may find the counterexample more quickly. The verification results with the register example (with multiple readers), shown in Table 2 ('Anti-Chain' and 'Without' are denoted as AC and W/o for short), evidences that anti-chain finds the counterexample more quickly in all three kinds of refinement checking. Nonetheless, we remark that because the algorithms are on-the-fly, whether the counterexample is found earlier depends on the searching order and sometimes anti-chain based algorithms may be slower if a 'wrong' order is taken.

4 Probabilistic Refinement Checking with Anti-chain

In this section, we show that anti-chain can be used to improve a particular kind of probabilistic refinement checking, i.e., the implementation is given as an MDP and the specification is given as an NFA.

4.1 MDP and Probabilistic Model Checking

Given a set of states S, a distribution is a function $\mu : S \to [0,1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S. A Markov Chain is a tuple $\mathcal{M} = (S, init, Act, Pr)$ where S is a countable set of states; $init \in S$ is an initial state; Act is a set of events; and $Pr : S \times Act \times S \to [0,1]$ is a labeled transition probability function such that for all state $s \in S$, $\exists\, e \in Act, \Sigma_{s' \in S} Pr(s,e,s') = 1$, and $\forall\, e' \neq$

$e, \Sigma_{s' \in S} Pr(s, e, s') = 0$. Notice that Markov Chains are deterministic as there is only one event (and one distribution) at each state.

A sequence of alternating states and events $\pi = \langle s_0, e_0, s_1, e_1, \cdots, e_n, s_{n+1} \rangle$ is a path of $\mathcal{M}$ if $Pr(s_i, e_i, s_{i+1}) > 0$ for all i. The probability of executing π from s_0, written as $Pr(\mathcal{M}, \pi)$, is $Pr(s_0, e_0, s_1) \times Pr(s_1, e_1, s_2) \times \cdots \times Pr(s_n, e_n, s_{n+1})$. It is often also interesting to find out the probability of reaching a certain set of states (e.g., what is the probability of reaching the state of system failure?). Given a set of target states G, the probability of reaching any state in G from a starting state s_0, written as $Pr(\mathcal{M}, s_0, G)$, is the accumulated probability of all paths from s_0 to any state in G, which can be calculated systematically [5]. Given a path π, we define $trace(\pi)$ to be the sequence of visible events in π. We write $Pr(\mathcal{M}, s_0, tr)$ to denote the probability of exhibiting a trace tr from state s_0, which is the accumulated probability of all paths π from s_0 such that $trace(\pi) = tr$. Given a set of traces Tr, the probability of $\mathcal{M}$ exhibiting any trace in Tr from state s_0 is the accumulated probability $\Sigma_{tr \in Tr} Pr(\mathcal{M}, s_0, tr)$.

Different from Markov Chains, an MDP can express both probabilistic choices and non-determinism. An MDP is a tuple $\mathcal{D} = (S, init, Act, Pr)$ where S is a set of system states; $init \in S$ is the initial system configuration[2]; Act is a set of actions; $Pr : (S \times Act) \to Distr(S)$ is a transition probability function such that for all states $s \in S$ and $a \in Act$: $\Sigma_{s' \in S} Pr(s, a, s') \in \{0, 1\}$. Notice that there could be multiple events at any state. A transition of the system is written as $s \xrightarrow{e} \mu$ where μ is a distribution. A path of $\mathcal{M}$ is a sequence of alternating states, events and distributions $\pi = \langle s_0, e_0, \mu_0, s_1, e_1, \mu_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} \mu_i$ and $\mu_i(s_{i+1}) > 0$ for all i. Given a path π, we define $trace(\pi)$ to be the sequence of visible events in π.

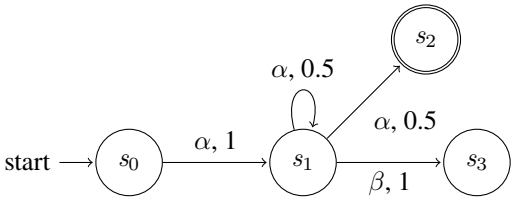
Intuitively speaking, given a system configuration, firstly an event and a distribution is selected non-deterministically by the *scheduler*, and then one of successor states is reached according to the probability distribution. A scheduler is a function deciding which event and distribution to choose based on the execution history. With a scheduler δ, we effectively obtain a Markov Chain from $\mathcal{D}$, written as $\mathcal{D}_\delta$. Note that with different scheduling, the probability of reaching a state or exhibiting a trace may be different. The measurement of interest is thus the maximum and minimum probability. Given a set of target states G and an MDP $\mathcal{D}$, the maximum probability of reaching any state in G from state s_0 is defined as $\mathcal{P}_{max}(\mathcal{D}, s_0, G) = \sup_\delta Pr(\mathcal{D}_\delta, s_0, G)$. Note that the supremum ranges over all, potentially infinitely many, schedulers. Accordingly, the minimum is written as $\mathcal{P}_{min}(\mathcal{D}, s_0, G)$. Similarly, we define the maximum probability of exhibiting a trace in a set Tr by $\mathcal{D}$ from s_0.

$$\mathcal{P}_{max}(\mathcal{D}, s_0, Tr) = \sup_\delta(\Sigma_{tr \in Tr} Pr(\mathcal{D}_\delta, s_0, tr))$$

Accordingly, the minimum is written as $\mathcal{P}_{min}(\mathcal{D}_\delta, s_0, Tr)$.

Example 4. The following shows a simple example MDP.

[2] This is a simplified definition. In general, there can be an initial distribution.



where s_0 is the initial state and s_2 is a target state. For simplicity, we omit the self-loop of s_2 and s_3. s_1 has two distributions, following two actions α and β. If s_1 non-deterministically chooses α, then it has equal probability to transfer to s_2 or stay in s_1; and if β is chosen, it will transfer to s_3 with probability 1. □

Definition 6 (Refinement Probability). *Let $\mathcal{D} = (S, init, Act, Pr)$ be an MDP; $\mathcal{L}$ be an LTS. The maximum probability of $\mathcal{D}$ trace-refining $\mathcal{L}$ is $\mathcal{P}_{max}(\mathcal{D}, init, traces(\mathcal{L}))$. The minimum is $\mathcal{P}_{min}(\mathcal{D}, init, traces(\mathcal{L}))$.*

Intuitively, the probability of $\mathcal{D}$ refines $\mathcal{L}$ is the probability of $\mathcal{D}$ exhibiting a trace of $\mathcal{L}$. As we mention earlier, the probability may vary due to different scheduling.

Definition 7 (MDP and LTS Synchronous Product). *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d)$ be an MDP; $\mathcal{L} = (S_l, init_l, Act_l, T)$ be an LTS such that $\tau \notin Act_l$. The synchronous product of $\mathcal{D}$ and $\mathcal{L}$, written as $\mathcal{D} \times \mathcal{L}$, is an MDP $(S, init, Act, Pr)$ such that $S = S_d \times S_l$; $init = (init_d, init_l)$; $Act = Act_d \cup Act_l$; and Pr is defined as follows.*

- *If $(s_1, \tau, \mu) \in Pr_d$, then $((s_1, s_2), \tau, \mu') \in Pr$ for all $s_2 \in S_l$ such that for all $s_1' \in S_d$, $\mu'((s_1', s_2)) = \mu(s_1')$;*
- *If $(s_1, e, \mu) \in Pr_d$ and $(s_2, e, s_2') \in T$, then $((s_1, s_2), e, \mu') \in Pr$ such that for all $s_1' \in S_d$, $\mu'((s_1', s_2')) = \mu(s_1')$.*

A state (s_1, s_2) of the product $\mathcal{D} \times \mathcal{L}$ is a TR-witness state if and only if $s_2 = \varnothing$. In [14], we show that the refinement probability can be calculated systematically by (1) building the deterministic LTS $det(\mathcal{L})$; (2) computing the synchronous product of $\mathcal{D}$ and $det(\mathcal{L})$; (3) calculating the maximum/minimum probability of reaching any TR-witness state.

Theorem 4. *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d)$ be an MDP; $\mathcal{L} = (S_l, init_l, Act_l, T)$. Let G be the set of TR-witness states of $\mathcal{D} \times det(\mathcal{L})$.*

- $\mathcal{P}_{max}(\mathcal{D}, init_d, traces(\mathcal{L})) = \mathcal{P}_{max}(\mathcal{D} \times det(\mathcal{L}), (init_d, init_l), G)$
- $\mathcal{P}_{min}(\mathcal{D}, init_d, traces(\mathcal{L})) = \mathcal{P}_{min}(\mathcal{D} \times det(\mathcal{L}), (init_d, init_l), G)$ □

Based on the above theorem, the probabilistic refinement checking problem is reduced to a probabilistic reachability problem, which can be solved by two standard methods. One is by solving a linear program. That is, we firstly associate a variable x_s to each state s in $\mathcal{P}$ to represent the probability of reaching any target state from s; then we construct a linear program which constraints the value of every x_s using a set of linear inequalities, based on the probability transition function; and lastly, we solve the linear program to get the maximum/minimum value of each x_s. Notice that the solution of state $(init_d, init_l)$ is the refinement probability. The other is to iteratively approximate the probability through graph traversing. Notice that for systems having large state

space, it is impractical to store the entire linear program and solve it directly, therefore the iterative calculation approach is more widely used in probabilistic verification. As a result, in this paper we just focus on this approach.

Example 5. In the following, we show how the iterative calculation method works using the simple example shown in Example 4. That is, the maximal probability from initial state s_0 to accepting state s_2 is calculated step by step. Assume p_i^k is the maximal probability of s_i after the k-th iteration. Starting from the target state s_2, in k-th iteration we update the probability of states which could reach s_2 in exact k steps. Obviously, $p_0^0 = p_1^0 = 0$. As $p_2^k = 1$ and $p_3^k = 0$ for any k, k is ignored in these two states. In the 1st iteration, only p_1 can be updated, and $p_1^1 = \max\{0.5 \times p_1^0 + 0.5 \times p_2^0, 1 \times p_3^0\} = max\{0.5, 0\} = 0.5.$; in the 2nd iteration, both p_0 and p_1 can be updated. It is trivial to show $p_0^2 = p_1^1 = 0.5$, and $p_1^2 = \max\{0.5 \times p_1^1 + 0.5 \times p_2^1, 1 \times p_3^1\} = max\{0.75, 0\} = 0.75$. Iteratively, p_0 and p_1 in the long run can be calculated. A user-defined threshold is usually necessary to terminate the calculation, according to the desired precision. □

4.2 Anti-chain Based Approach

Now we introduce how anti-chain can be used to speed up the iterative calculation approach, by first introducing a lemma.

Lemma 3. *Let $\mathcal{D} = (S_d, init_d, Act_d, Pr_d)$ be an MDP; $\mathcal{L} = (S_l, init_l, Act_l, T)$. Let $\mathcal{P}$ be $\mathcal{D} \times det(\mathcal{L})$. Let G be the set of TR-witness states of $\mathcal{P}$. For all state (u_1, v_1) and (u_2, v_2) of $\mathcal{P}$ s.t. $(u_2, v_2) \leq (u_1, v_1)$, $Pr_{max}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{max}(\mathcal{P}, (u_2, v_2), G)$ and $Pr_{min}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{min}(\mathcal{P}, (u_2, v_2), G)$.*

Proof The above can be proved with an induction. The base case is that (u_2, v_2) is in G. By definition, (u_1, v_1) must be in G and therefore the lemma holds. Next, we show the induction step. Assume that (u_2', v_2') satisfies the lemma above. For every distribution μ_2 from (u_2, v_2), by Definition 7, there must exist a distribution μ_1 from (u_1, v_1) and for every state (u_2', v_2'), there exists (u_1', v_1') such that $\mu_2((u_2', v_2')) = \mu_1((u_1', v_1'))$ and $(u_2', v_2') \leq (u_1', v_1')$. By induction hypothesis, we have $Pr_{max}(\mathcal{P}, (u_1', v_1'), G) \geq Pr_{max}(\mathcal{P}, (u_2', v_2'), G)$ and $Pr_{min}(\mathcal{P}, (u_1', v_1'), G) \geq Pr_{min}(\mathcal{P}, (u_2', v_2'), G)$. Thus we have $Pr_{max}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{max}(\mathcal{P}, (u_2, v_2), G)$ and $Pr_{min}(\mathcal{P}, (u_1, v_1), G) \geq Pr_{min}(\mathcal{P}, (u_2, v_2), G)$. Therefore, we conclude that the lemma holds. □

Compared to probabilistic reachability calculation for a general MDP, the above lemma gives us additional information, which can be potentially useful in speeding up the calculation. In the following, we discuss how we can make use of the information so as to improve the probabilistic refinement checking using the iterative calculation method.

The first step is building the product MDP meanwhile finding the target states, which is shown in Algorithm 4. The implementation and specification are defined in Definition 7. Different from the non-probabilistic cases, *the state space cannot be reduced in the probabilistic models*; instead, we define a function sub of the product state s satisfying $s.sub = \{t \mid t \in S \wedge s \leq t\}$, where S is the state space of the product MDP. Then the refinement checking is reduced to probabilistic reachability of a set of target states, denoted by $Target$. During the iterative calculation, whenever the probability of

Algorithm 4 Building MDP in Probabilistic Refinement Checking with Anti-chain

```
1: let working be a stack containing a pair (init_d, {s | init_l ⇝ s});
2: let visited := {(init_d, {s | init_l ⇝ s})}; let Target = ∅; ;
3: while working ≠ ∅ do
4:     pop (impl, spec) from working;
5:     for all (impl, e, μ) ∈ Pr_d do
6:         if e = τ then
7:             spec' := spec;
8:         else
9:             spec' := {s' | ∃ s ∈ spec. s -e-> s'};
10:        for all impl' ∈ S_d do
11:            if μ(impl') > 0 ∧ (impl', spec') ∉ visited then
12:                push (impl', spec') into working;
13:                visited := visited ∪ (impl', spec');
14:                if spec' = ∅ then
15:                    Target := Target ∪ (impl', spec');
16:                for all (impl', spec'') ∈ visited do
17:                    if (impl', spec'') ≤ (impl', spec') then
18:                        (impl', spec'').sub.Add(impl', spec');
19:                    else if (impl', spec') ≤ (impl', spec'') then
20:                        (impl', spec').sub.Add(impl', spec'');
21: return true;
```

state s is updated, e.g., to p, according to lemma 3, **all states in** $s.sub$ **whose probability is less than** p **could be set to** p **directly.** This could speed up each iteration and potentially improve probabilistic refinement checking.

Now we evaluate whether the above method is indeed beneficial. The proposed probabilistic refinement checking algorithm has also been implemented in PAT. We evaluate it using a modified system based on the implementation of a distributed concurrent stack example [15]. Probabilistic choices are used to model a concurrent stack model composed by *two processes*, so as to capture the situation in which the communication between different processes fails from time to time. Failures do exist in real world cases and the experiments results are summarized in Table 3.

Table 3. Experiments: Probabilistic Concurrent Stack Implementation

System	Size	Verification Time (s)			#States Involved in Iterations		
		W/o AC	With AC	Gain	W/o AC	With AC	Gain
K = 2	20600	2.74	2.21	19.3%	4.2M	3M	28.6%
K = 3	45584	15.98	12.04	24.6%	18.6M	11.7M	37.1%
K = 4	86704	48.72	37.50	22.6%	55.5M	36.2M	34.8%
K = 5	117408	123.9	80.83	34.9%	130.7M	76.3M	41.6%
K = 6	231440	271.2	182.6	32.7%	272.1M	160.7M	40.9%
K = 7	342544	511.1	340.3	33.5%	515.2M	298.8M	42.0%

We compare the efficiency of the implementation with and without (W/o) Anti-chain (AC) using several cases. K means length of the stack; $Size$ indicates the number of states in the whole system; *#States Involved in Iterations* represents the total number of states involved in the iterative calculation. For example, a state s updates its probability in two iterations, then *#States* should increase two. From the experiments, we can see that the anti-chain approach could reduce the total number of states accumulated during the calculation, through dynamically updating states' probability based on the subset relation sub. This speeds up the verification around 29%. We remark that the gains here are not as significant as the non-probabilistic cases, because the state space cannot be reduced; however, in some cases, it does shorten the verification time.

5 Conclusion

In this work, we proposed to adopt anti-chain approach to improve stable failures refinement, failures-divergence refinement and probabilistic refinement checking. These algorithms have been implemented in model checking framework PAT, and some experiments based on benchmark systems demonstrated the dramatic improvement of the verification efficiency of our method. To our best of knowledge, we are the first to investigate anti-chain approaches for these refinement checking.

As for future work, we are trying to extend anti-chain based refinement checking approach in real-time system; meanwhile, we are exploring the refinement relation between probabilistic models, which may also benefit from anti-chain based method.

References

1. Roscoe, A.W.: Model-checking CSP, ch. 21. Prentice-Hall (1994)
2. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
3. Aguilera, M.K., Gafni, E., Lamport, L.: The Mailbox Problem. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 1–15. Springer, Heidelberg (2008)
4. Attiya, H., Welch, J.: Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd edn. The Oxford University Press (2004)
5. Baier, C., Katoen, J.: Principles of Model Checking. The MIT Press (2008)
6. Bouajjani, A., Habermehl, P., Holík, L., Touili, T., Vojnar, T.: Antichain-Based Universality and Inclusion Testing over Nondeterministic Finite Tree Automata. In: Ibarra, O.H., Ravikumar, B. (eds.) CIAA 2008. LNCS, vol. 5148, pp. 57–67. Springer, Heidelberg (2008)
7. Chatterjee, K., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Algorithms for Omega-Regular Games with Imperfect Information. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 287–302. Springer, Heidelberg (2006)
8. Doyen, L., Raskin, J.F.: Antichains for the automata-based approach to model checking. Logical Methods in Computer Science 5(1:5), 1–20 (2009)
9. Ellen, F., Lev, Y., Luchangco, V., Moir, M.: SNZI: Scalable nonzero indicators. In: PODC, pp. 13–22. ACM (2007)
10. Filiot, E., Jin, N., Raskin, J.-F.: An Antichain Algorithm for LTL Realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)

11. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)
12. Roscoe, A.W.: On the expressive power of CSP refinement. Formal Aspects of Computing 17(2), 93–112 (2005)
13. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
14. Sun, J., Song, S., Liu, Y.: Model Checking Hierarchical Probabilistic Systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 388–403. Springer, Heidelberg (2010)
15. Treiber, R.K.: Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center (1986)
16. De Wulf, M., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
17. De Wulf, M., Doyen, L., Maquet, N., Raskin, J.-F.: Antichains: Alternative Algorithms for LTL Satisfiability and Model-Checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 63–77. Springer, Heidelberg (2008)

An Analytical and Experimental Comparison of CSP Extensions and Tools

Ling Shi[1], Yang Liu[2], Jun Sun[3], Jin Song Dong[1], and Gustavo Carvalho[4]

[1] SoC, National Univ. of Singapore
{shiling,dongjs}@comp.nus.edu.sg
[2] Temasek Lab, National Univ. of Singapore
tslliuya@nus.edu.sg
[3] ISTD, Singapore Univ. of Technology and Design
sunjun@sutd.edu.sg
[4] Centro de Informática, UFPE, Brazil
ghpc@cin.ufpe.br

Abstract. Communicating Sequential Processes (CSP) has been widely applied to modeling and analyzing concurrent systems. There have been considerable efforts on enhancing CSP by taking data and other system aspects into account. For instance, CSP$_M$ combines CSP with a functional programming language whereas CSP# integrates high-level CSP-like process operators with low-level procedure code. Little work has been done to systematically compare these CSP extensions, which may have subtle and substantial differences. In this paper, we compare CSP$_M$ and CSP# not only on their syntax, but also operational semantics as well as their supporting tools such as FDR, ProB, and PAT. We conduct extensive experiments to compare the performance of these tools in different settings. Our comparison can be used to guide users to choose the appropriate CSP extension and verification tool based on the system characteristics.

1 Introduction

Communicating Sequential Processes (CSP) [3], a prominent member of the process algebra family, has been designed to formally model *concurrent systems*. It represents system behavior in terms of processes constituted by a rich set of compositional operators. CSP also provides algebraic laws such that equivalence of process expressions can be rigorously established. It has been applied to a variety of safety-critical systems [25].

With the increasing size and complexity of concurrent systems, it becomes clear that CSP has its limitations in modeling systems with non-trivial data structures (e.g., *array*) or functional aspects. To solve this problem, many considerable efforts on enhancing CSP have been made. The most noticeable is CSP$_M$ [15], a machine-readable dialect of CSP, combining CSP with a *functional* programming language. Recently, CSP# (Communicating Sequential Programs) [22] has been proposed to integrate high-level CSP-like process operators with low-level program constructs such as *assignments* and *while* loops. Although these languages support CSP-like modeling notations and can deal with similar types of concurrent systems, there are subtle and substantial differences between them. For example, concurrency is captured differently; CSP$_M$

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 381–397, 2012.

only supports synchronous channel communications, while CSP# supports both synchronous/asynchronous channels and shared variables. In addition, those differences can lead to different verification capabilities empowered by their respective analysis tools, i.e., FDR (Failures Divergence Refinement) [10] and ProB [6] for CSP$_M$, and PAT (Process Analysis Toolkit) [23] for CSP#. Little work has been conducted to provide a comprehensive investigation of these CSP extensions so as to help users choose appropriate languages/tools for various systems from the perspectives of modeling and verification needs.

In this work, we systematically compare CSP$_M$ and CSP# in terms of three aspects, i.e., language syntax, operational semantics, and reasoning power of their supporting tools. Firstly, we show the syntactic differences, followed by comparing the operational semantics. We also discuss the transformation between CSP$_M$ and CSP# models. Secondly, we characterize various reasoning techniques and verifiable properties of FDR, ProB and PAT, respectively. Next, we explore the strengths and limits of the languages and tools by modeling and verifying nine systems, each of which is designed to show particular features of the languages or the tools. Lastly, we investigate reasons behind the experiment results; particularly, the semantic differences between CSP$_M$ and CSP# lead to different state spaces and optimizations in model checking.

We believe that the comparison is useful for the following reasons. Firstly, our comparison may guide users to select an appropriate modeling language. The decision depends on system features (e.g., shared variables, etc.) and properties to prove (e.g., compositional refinement checking, etc.). Secondly, our analysis of languages that are designed for concurrent systems in terms of simplicity and expressiveness (e.g., communications via channels or shared memory) can act as a reference in designing new programming languages of concurrent systems. Thirdly, the translation discussed in the paper can help users to change their models between CSP$_M$ and CSP#, and hence to utilize different reasoning power of their respective reasoning tools. Lastly, our experiments with FDR, ProB, and PAT provide qualitative analysis of tool capability/efficiency.

2 CSP$_M$ vs. CSP#: Syntax

CSP$_M$ enriches CSP with an expression language that is based on *functional* foundations. It mainly uses event synchronization to specify concurrent systems, and supports operators like linked parallel $P[c <->> c']Q$ in which two different channels c and c' from processes P and Q respectively run synchronously. CSP# not only inherits event synchronization and compositional process constructs from CSP, but also supports additional features like asynchronous channel communication, imperative programs, etc. In this section, we elaborate the differences between these two languages in terms of their syntax. Table 1 shows common CSP, CSP$_M$ and CSP# process definitions, where P (and Q) is a process with an optional list of parameters; a is an event name; A and A' are sets of event names and channel expressions; b is a Boolean expression; c and c' are channel names; e is an expression; x and x' are variables; and V is a set of accepted values. We illustrate the detailed differences from two perspectives.

Data Perspective CSP$_M$ supports functional paradigm, where process parameters can take in processes, functions, and channels. This is not available in CSP# which adopts

Table 1. Similar Syntax among CSP, CSP$_M$ and CSP#

CSP	**CSP$_M$**	**CSP#**	**Description**
$STOP$	$STOP$	$Stop$	deadlock
$SKIP$	$SKIP$	$Skip$	termination
$CHAOS$	$CHAOS(A)$	-	chaotic process
$a \rightarrow P$	$a \rightarrow P$	$a \rightarrow P$	event prefixing
$c!e \rightarrow P$ $c?x \rightarrow P$	$c?x?x' : V!e \rightarrow P$	$c!e \rightarrow P$ $c?[b]x \rightarrow P$	channel communication
$P \Box Q$	$P \,[]\, Q$	$P \,[*]\, Q$	external choice
$P \sqcap Q$	$P \,\|\sim\|\, Q$	$P <> Q$	internal choice
$P;\ Q$	$P;\ Q$	$P;\ Q$	sequential composition
$P \setminus A$	$P \setminus A$	$P \setminus A$	hiding
$x := e$	-	$x := e$	assignment
$P \lhd b \rhd Q$	$if\ b\ then\ P\ else\ Q$	$if\ b\ then\ P\ else\ Q$	conditional choice
$P \parallel Q$	$P[\|\ A\ \|]Q$ $P[A \parallel A']Q$ $P[c <-> c']Q$	$P \parallel Q$	parallel composition
$P \,\|\|\|\, Q$	$P \,\|\|\|\, Q$	$P \,\|\|\|\, Q$	interleaving
$P \triangle Q$	$P/\backslash Q$	$P\ interrupt\ Q$	interrupt

imperative paradigm, although this limitation may be resolved partially through 'clever' modeling. For instance, a CSP$_M$ concrete process $System = P(Sys1, Sys2)$ associated with an abstract process $P(P1, P2) = a \rightarrow P1 \,[]\, b \rightarrow P2$ can be translated to a CSP# concrete process $System = a \rightarrow Sys1 \,[*]\, b \rightarrow Sys2$, here $Sys1$ and $Sys2$ are processes. However, it may not be possible to specify abstract process behavior (e.g., process P in this example) in CSP#, whose parameters are processes.

CSP$_M$ enables rich data expressions such as sequences, sets, Boolean, tuples, and lambda calculus. It also allows users to define data types using the reserved word "datatype". CSP# directly supports integers, Boolean, array of integers or Boolean. In addition, it supports user-defined data types and corresponding operations using imperative languages like C#[1], C, or Java. Functions can be declared in CSP$_M$ following the functional paradigm, while in CSP#, they are encoded as processes or defined as static C# methods (which can be invoked via method *call* in CSP# models).

A channel in CSP$_M$ is declared with an explicit type. Values communicated through a channel must be in their type range; otherwise, an error is reported at run time by FDR and ProB. Moreover, CSP$_M$ is dynamically typed in FDR, namely, there is no way to declare the types of functions and variables (process parameters), while ProB can type check the CSP$_M$ models in a dynamic or (optional) static way [7]. In contrast, CSP# is weak typed (a.k.a. loose typing) and therefore no type information is required when declaring a variable or channel. Channels are declared with its name and buffer size. If the buffer size is 0, then it is declared as a synchronous channel, otherwise it is an asynchronous channel. The process parameters and channel input variables can take in values with different types at different time. As long as there is no type mismatch (e.g.,

[1] C# is the best supported language in PAT and used as the representative language in this paper.

using an integer as a guard condition), the execution can proceed; otherwise, invalid type casting exception is raised at run time.

Process Perspective. One big difference is that CSP# directly supports shared variables. Unlike CSP$_M$ which excludes assignments of shared variables [10], CSP# treats assignments as an important modeling feature. In CSP#, an event can be associated with an imperative program, which is executed *atomically* together with the occurrence of the event. For instance, an event associated with a program (referred to as a data operation) is written as $a\{prog\} \rightarrow P$ where $prog$ is the program and a is an event name. We remark that a shared variable can be modeled as a process parallel to the one that uses the variable (see [3] and [17]). Recently, shared variable analyzer (SVA) [17], a front-end of FDR, has been developed to convert programs (like C programs) with shared variables into CSP$_M$ models, in which shared variables are modeled as *variable processes*; reading from/writing to those shared variables are carried out over channels. We illustrate the modeling of shared variables in Section 3.

Asynchronous channels, as a popular and practical type of communication mechanism for networked systems, are directly supported in CSP#. Given an asynchronous channel ac with a positive buffer size, $ac!e \rightarrow P$ evaluates expression e with the current variable valuation, puts the value into the tail of the respective buffer for ac and then behaves as P. In contrast, $ac?x \rightarrow P$ (and $ac?[b]x \rightarrow P$) gets the top element from the respective buffer, assigns it to variable x and then behaves as P (the latter further constrains the received data to satisfy the Boolean condition b). Buffers store messages in a first-in-first-out (FIFO) order. Notice that asynchronous channels in CSP# are similar to those supported in Promela [4]. Although asynchronous channels are not directly supported in CSP$_M$, they can be modeled as buffer processes by event synchronization, which will be shown in Section 3.

In CSP$_M$, users are required to indicate synchronized events in three kinds of parallel compositions, which are, *sharing* ($P[|\ A\ |]Q$), *alphabetized parallel* ($P[A\ \|\ A']Q$), and *linked parallel* ($P[c \leftrightarrow c']Q$). On the other hand, CSP# supports only alphabetized parallel composition and frees users from specifying explicit alphabets of processes in parallel; a sophisticated procedure [22] calculates automatically a *default* alphabet of a process which is the set of events that constitute the process expression. Nevertheless, this procedure may not work when an event name consists of global variables or process parameters which change through *recursive calls*; in such a case, users need to specify the alphabet of a process. Notice that in order to avoid data race, data operations are not a part of the alphabet and therefore are never synchronized.

In CSP#, an event can have the name tau to represent the invisible event τ in event prefixing or data operations, e.g., $tau \rightarrow Stop$ or $tau\{prog\} \rightarrow Stop$. With the support of tau event, users can avoid using hiding operator to explicitly hide some visible events by naming them tau. *External* and *internal* choices are supported in both languages. Moreover, CSP# allows *general choice* $P[]Q$ in which the choice is resolved by any event. This operator is more like the CCS + operator, which can be resolved by a τ event performed by either process. Nonetheless, the general choice operator can be simulated in CSP$_M$ [14].

Besides the common conditional choice, CSP# copes with two additional types of conditional choices to facilitate modeling: *atomic* conditional choice $ifa\ b\ \{P\}\ else\ \{Q\}$

and *blocking* conditional choice $\mathit{if}b\ b\ \{P\}$. With the former, the checking of condition b is to be conducted *atomically* with the occurrence of the first event in P or Q. The latter is blocked when b is unsatisfied.

Both CSP$_M$ and CSP# define Boolean guard $b\&P$ and $[b]P$ respectively; process waits until condition b becomes true and then behaves as P. Replicated process operators, such as replicated external/internal choices, replicated parallel and interleaving, are also supported in both languages. Chaotic process ($\mathit{CHAOS}(A)$), event renaming ($P[[c \leftarrow c']]$), and untimed timeout ($P[> Q$) defined in CSP$_M$ are not directly handled in CSP#. We discuss how to model these features using CSP# operators in Section 3.

So far we have shown the syntactic differences between CSP$_M$ and CSP#. Both CSP$_M$ and CSP# support dedicated syntax which is unavailable in the other. Some special syntax operators in one can be indirectly achieved in the other. For instance, the *CHAOS* process in CSP$_M$ can be defined in CSP# using choices and event prefixing (discussed in the next section). Nonetheless, it is not always trivial to support some dedicated syntax operators such as shared variables in CSP$_M$ and channel communications in CSP# (which can involve multiple processes).

3 CSP$_M$ vs. CSP#: Operational Semantics

Operational semantics describes the sequences of computational steps that a model can take. We illustrate the operational semantics of CSP$_M$ and CSP# in the form of labeled transition systems (LTS). An LTS is 3-tuple $\mathcal{L} = (S, \mathit{init}, \rightarrow)$ where S is a set of system configurations; $\mathit{init} \in S$ is an initial system configuration and $\rightarrow$: $S \times \Sigma \cup \{\checkmark, \tau\} \times S$ is a labeled transition relation. Note that $\Sigma \cup \{\checkmark, \tau\}$ is the event space where Σ is the set of *visible* events, $\checkmark$ denotes a successful termination, and τ is an *invisible* event.

A system configuration S in CSP$_M$ is a *pair* of processes and environment where the latter maps variable identifiers to values such as data, processes, or a distinguished *error* configuration. In CSP#, S is composed of two components (V, P) where V maps variable names (or channel names) to values (or sequences of items in buffers), and P is a process expression. The operational semantics of a process construct is depicted by associated firing rule(s). CSP$_M$ and CSP# share very similar firing rules for some process constructs like *interrupt* [15,19,22]. We elaborate subtle differences in the operational semantics of six process constructs; the complete description of *all* different process constructs can be found in our technical report [21]. Note that CSP$_M$ process constructs like P in the firing rules below include the environment, same as [15].

SKIP Process SKIP means termination; namely, $\checkmark$ takes place followed by doing nothing, as captured by Stop in CSP#, whereas this is denoted by a special process term Ω in CSP$_M$. For simplicity, we use prefix M to refer to CSP$_M$ firing rules (e.g., M_skip), and # for CSP# (e.g., $\#\_\mathit{skip}$) in the following.

$$\frac{}{\mathit{SKIP} \xrightarrow{\checkmark} \Omega}\ [\ M\_\mathit{skip}\ ] \qquad \frac{}{(V, \mathit{Skip}) \xrightarrow{\checkmark} (V, \mathit{Stop})}\ [\ \#\_\mathit{skip}\ ]$$

Notice that in both CSP$_M$ and CSP#, $\checkmark$ may only be the last event of a trace. The semantic difference shown above thus will not result in different verification results in

FDR, ProB and PAT[2]. Nonetheless, it should be noticed that this difference leads to a different semantics for parallel composition as we show later.

CHAOS Process *CHAOS* in CSP$_M$ denotes the most non-deterministic process.

$$\frac{}{\mathit{CHAOS}(A) \xrightarrow{\tau} \mathit{STOP}}\,[\ M\_c1\ ] \qquad \frac{a \in A}{\mathit{CHAOS}(A) \xrightarrow{a} \mathit{CHAOS}(A)}\,[\ M\_c2\ ]$$

$\mathit{CHAOS}(A)$ is not directly supported by CSP# because of two main reasons. First, users have to specify all the events in set A to model *CHAOS*, whereas CSP# is designed to free users from specifying events associated with processes (if possible). Second, *CHAOS* is more useful in the failures/divergence checking, whereas CSP# models focus more on states/LTL checking. $\mathit{CHAOS}(A)$ can be manually captured in CSP# by constructing an equivalent process including all events. For example, let set A contains events a and b, one way to model $\mathit{CHAOS}(A)$ process in CSP# can be as follows.

$$\mathit{CHAOS\_A} = \mathit{tau} \rightarrow \mathit{Stop}\ [\!]\ a \rightarrow \mathit{CHAOS\_A}\ [\!]\ b \rightarrow \mathit{CHAOS\_A}$$

Channel communication. Channel communications are crucial in concurrent systems and they are classified into two types: *synchronous* and *asynchronous*. CSP$_M$ directly supports the former, whereas CSP# supports both. Both languages have their own operational semantics to interpret channel communications, which is elaborated below. The transformation of channel communication between CSP$_M$ and CSP# is discussed later.

A general format to express a channel communication is $cf \rightarrow P$, where c is a channel name, f a sequence of communication fields, and P a process with the scope of the prefix. A communication field can be an output (by $!e$ where e is an expression), an unconstrained input (by $?x$ where x is a variable), or a constrained input (by $?x : V$ in CSP$_M$ where V is a value range, and by $?[b]x$ in CSP# where b is a Boolean condition).

In CSP$_M$, channels are synchronous and communications are achieved by means of event synchronization. Specifically, assume the type of data communicated over channel c is T, $c!e \rightarrow P$ outputs a communication $c.v$ where v is the value of e and $v \in T$, and $c?x \rightarrow P$ accepts an input of the form $\{c.v \mid v \in T\}$; $c?x : V \rightarrow P$ imposes an additional constraint for $c.v$, namely, $v \in V$. As a channel can be associated with a sequence of communication fields in CSP$_M$, multi-part communications involving multiple data transfers can occur within a single action. For instance, $c?x : V!e \rightarrow P$ engages communications of the form $\{c.v'.v \mid v'.v \in T \wedge v' \in V\}$ where v is a value of e. The firing rule of the CSP$_M$ channel communication is presented below, where function $\mathit{comms}(cf)$ returns the set of communications described by cf and function $\mathit{subs}(a, cf, P)$ returns a process whose identifier in process P bounded by cf is substituted by event a.

$$\frac{a \in \mathit{comms}(cf)}{cf \rightarrow P \xrightarrow{a} \mathit{subs}(a, cf, P)}\,[\ M\_com\ ]$$

[2] Except deadlock-freeness checking; namely, a process is deadlock free *iff* it satisfies the deadlock-freeness assertion in FDR and ProB, whereas it has to satisfy both deadlock-freeness and nontermination assertions in PAT.

In CSP#, a channel is defined as a buffer which stores messages in a first-in-first-out (FIFO) order. Channels are synchronous when their buffer sizes are zero, in which case communications are realized by the hand shaking mechanism. Channels are asynchronous when their buffer sizes are bigger than zero, and their communications are achieved by the message passing mechanism. Sending and receiving multiple messages at one time are supported in both synchronous and asynchronous communications. In addition, the data type of the messages are *untyped* in CSP#. We show below the firing rules of CSP# for channel communications.

– A synchronous communication occurs when both processes $c!e \rightarrow P$ and $c?x \rightarrow P$ (or $c?[b]x \rightarrow P$) can be executed *simultaneously* and the messages passed match (and condition b is true); event $c.v$ is transferred where v is the value of e with the *latest* valuation $eva(V, e)$. In the following firing rule which is associated with parallel composition (the case for interleaving is similar), process $Q[eva(V, e)/x]$ replaces x with the new value v.

$$\frac{(V, c!e \rightarrow P) \overset{c!eva(V,e)}{\rightarrow} (V, P),\ (V, c?[b]x \rightarrow Q) \overset{c?[b]x}{\rightarrow} (V, Q),\ (V \wedge x = eva(V, e)) \Rightarrow b}{(V, c!e \rightarrow P \parallel c?[b]x \rightarrow Q) \overset{c.eva(V,e)}{\rightarrow} (V, P \parallel Q[eva(V, e)/x])}\ [\ \#\_par1\ ]$$

– An output process $ac!e \rightarrow P$, where ac is an asynchronous channel, is enabled if the associated buffer is not full. The process first evaluates e and then pushes the value into the tail of respective buffer for ac (denoted by function $app(V, ac!e)$), followed by the execution of P.

$$\frac{ac \text{ is not full in } V}{(V, ac!e \rightarrow P) \overset{ac!eva(V,e)}{\rightarrow} (app(V, ac!e), P)}\ [\ \#\_out\ ]$$

– A constrained input process $ac?[b]x \rightarrow P$ is enabled if the associated buffer size is not empty and b is valid with the latest valuation (denoted by function $top(ac)$). The process pops (denoted by function $pop(V, ac?x)$) and assigns the top element from the buffer to x, followed by the execution of P. Note that the checking of b is unnecessary for an unconstrained input process.

$$\frac{ac \text{ is not empty in } V \ \wedge (V \wedge x = top(ac)) \Rightarrow b}{(V, ac?[b]x \rightarrow P) \overset{ac?top(ac)}{\rightarrow} (pop(V, ac?x), P[top(ac)/x])}\ [\ \#\_in\ ]$$

Example 1. We exemplify below how CSP# captures CSP$_M$ multi-part synchronous channels and how CSP# asynchronous channels are represented in CSP$_M$. The event-like channel communication in CSP$_M$ can be modeled as alphabetized event-based synchronization in CSP#. We capture the channel communication by expanding the channel values according the type values. Specifically, an output process $c!e \rightarrow P$ is translated to a process $c.e \rightarrow P$ in CSP#, and an input process is transformed into a CSP# model

which enumerates *all* possible communications using the general choices ([]) to combine relevant event prefixing processes. Taking the following CSP$_M$ model of a vending machine (VM) as an example,

1. $datatype\ Drink = Sprite \mid Coke \mid Tea \mid Coffee$
2. $channel\ offer : Drink$
3. $VM = offer?x : diff(Drink, \{Coffee\}) \rightarrow VM$

where process VM can perform any communication in the form $\{offer.x \mid x \in diff(Drink, \{Coffee\}) \wedge x \in Drink\}$; function $diff(Drink, \{Coffee\})$ restricts that a vending machine can offer any drink except coffee. This VM can be captured by the following CSP# process where all possible communications are explicitly specified.

$$VM = offer.Sprite \rightarrow VM \ [] \ offer.Coke \rightarrow VM \ [] \ offer.Tea \rightarrow VM$$

An asynchronous channel in CSP# can be modeled as a CSP$_M$ process which represents the FIFO buffer by sending/receiving messages to/from other processes. We provide such a CSP$_M$ process below, where a sequence is defined in process $Buffer$ to store the message in the FIFO order and rcv and snd are channels.

1. $Buffer(c, \langle\rangle, N) = rcv?c?x \rightarrow Buffer(c, \langle x\rangle, N)$
2. $Buffer(c, s \frown \langle a\rangle, N) = \#s < N - 1 \& rcv?c?x \rightarrow Buffer(c, \langle x\rangle \frown s \frown \langle a\rangle, N)$
3. $\qquad\qquad\qquad\qquad [] \ snd!c!a \rightarrow Buffer(c, s, N)$

In the above $Buffer$ process, line 1 describes the situation where the buffer is empty, namely, only receiving messages from other process is allowed. Lines 2 and 3 depict message receiving and sending when the buffer is not full. This $Buffer$ process can be used to run in parallel with other process, say P, to perform asynchronous channel communication; for instance, a communication over an asynchronous channel ac with buffer size 2 can be modeled as $P[snd \leftrightarrow rcv, rcv \leftrightarrow snd]Buffer(ac, \langle\rangle, 2)$. We remark that asynchronous channel can be regarded as a special kind of shared variable, which is discussed in the next section; the way that asynchronous channels are modeled in CSP$_M$ is similar to handle shared variables in CSP$_M$ later.

Shared variables. Shared variables are important in modeling shared resources. Variables in Hoare's CSP processes are local and disjoint. We elaborate below how shared variables are supported by CSP# directly and CSP$_M$ indirectly.

CSP# uses shared variables to model data states and operations in a procedural style. The operations are modeled as terminating sequential programs in the form $a\{prog\} \rightarrow P$, where programs $prog$ can contain local variables[3], if-then-else statements, while loops, the invocation of external libraries written in C#/Java (through the *reflection* techniques). The execution of the programs is atomic together with the occurrence of associated events. In the following firing rules, function $upd(V, prog)$ returns a modified valuation function according to the particular semantics of the program; in $prog$, both shared and local variables can be used and updated.

$$\frac{}{(V, a\{prog\}[4] \rightarrow P) \xrightarrow{a} (upd(V, prog), P)} [\ \#\_dataOp\ ]$$

[3] The scope of local variables is within $prog$, and they are not stored in valuation function V.

[4] Event a can also be an invisible event, denoted as tau, then the transition event becomes τ.

Shared variables can be modeled in CSP$_M$ indirectly as discussed in [17]. To be specific, a shared variable is represented by a *variable process* which is executed concurrently with other *user processes* which invoke the variable. Variable processes are modeled as read/write operations, and hence user processes can read from/write to the shared variables by CSP$_M$ synchronous communication. For example, the following processes $Var(v, val)$ and $Var\_A(j, v, val)$ execute together as a variable process to denote a shared variable v, where val is the value of v and j denotes a unique id of a user process which invokes v. The constraint that only one process is allowed to read/write v is specified in Var_A which is triggered by event $start\_at?j!v$ from Var.

1. $Var(v,\ val)\ =\ read?i!v!val\ \rightarrow\ Var(v,\ val)$
2. $\quad [] \ write?i!v?x\ \rightarrow\ Var(v,\ x)\ []\ start\_at?j!v\ \rightarrow\ Var\_A(j,\ v,\ val)$
3. $Var\_A(j,\ v,\ val)\ =\ read.j!v!val\ \rightarrow\ Var\_A(j,\ v,\ val)$
4. $\quad [] \ write.j!v?x\ \rightarrow\ Var\_A(j,\ v,\ x)\ []\ end\_at?j!v\ \rightarrow\ Var(v,\ val)$

Example 2. The following CSP# model and CSP$_M$ model represent the same system which sums three process parameters, where the processes are selected nondeterministically from three processes. In the CSP# model below, sum and $count$ are shared variables with initial value 0, and their updates are executed atomically with the occurrence of event add in process $P(i)$.

1. $var\ count\ =\ 0;\quad var\ sum\ =\ 0;$
2. $P(i)\ =\ [count\ <\ 3]add\{sum = sum + i;\ \ count = count + 1;\ \}\ \rightarrow\ P(i);$
3. $System()\ =\ |||\ i : \{1..3\}@P(i);$

In the CSP$_M$ model, the shared variables sum and $count$ are modeled as variable processes $Var(sum, 0)$ and $Var(count, 0)$. In addition, process $P(i)$ is defined (lines 2 to 4) by a sequence of variable access events (e.g., events $start\_at!i!count$ and $end\_at!i!count$ for $count$).

1. $datatype\ VarDt\ =\ count\ |\ sum\quad T = \{1..3\}\ \ Range\ =\ \{0..10\}$
2. $P(i)\ =\ start\_at!i!count\ \rightarrow\ read!i?count?x\ \rightarrow\ x < 3\ \&\ add$
3. $\qquad \rightarrow\ start\_at!i!sum\ \rightarrow\ read!i?sum?y\ \rightarrow\ write!i!sum!(y + i)$
4. $\qquad \rightarrow\ write!i!count!(x + 1)\ \rightarrow\ end\_at!i!sum\ \rightarrow\ end\_at!i!count\ \rightarrow\ P(i)$
5. $Processes()\ =\ |||\ i : \{1..3\}@P(i)$
6. $Variables()\ =\ Var(count,\ 0)\ |||\ Var(sum,\ 0)$
7. $SharedEvent\ =\ \{read.t.v.val,\ write.t.v.val,\ start\_at.t.v,\ end\_at.t.v\ |$
8. $\qquad\qquad t \leftarrow T,\ v \leftarrow VarDt,\ val \leftarrow Range\}$
9. $System()\ =\ Variables()\ [|\ SharedEvent\ |]\ Processes()$

As shown above, CSP# allows users to specify shared variables and their operations in a way similar to imperative programming languages, which allows users to see variable states at each simulation step. In contrast, CSP$_M$ supports shared variables by the means of auxiliary processes and events; the additional operations may result in more system states during model checking, as shown later in our experiments.

Parallel composition. The firing rules of parallel composition $P\ \|\ Q$ in CSP$_M$ and CSP# are similar except the way of handling the ✓ event. Both languages require

distributed termination: process $P \parallel Q$ terminates if both P and Q terminate. This requirement is satisfied in CSP# by the following firing rule.

$$\frac{(V,P) \xrightarrow{\checkmark} (V,P'), (V,Q) \xrightarrow{\checkmark} (V,Q')}{(V,P \parallel Q) \xrightarrow{\checkmark} (V,Stop)}\,[\,\#\_par2\,]$$

In addition, CSP$_M$ allows the termination of a paralleled process to be independent of its associated process. Firing rules $[M\_par1]$ below describes that the termination of P involves an invisible event τ and P becomes Ω; operator $\underset{X}{\parallel}$ is a general form of three kinds of parallel operators in CSP$_M$.

$$\frac{P \xrightarrow{\checkmark} P'}{P \underset{X}{\parallel} Q \xrightarrow{\tau} \Omega \underset{X}{\parallel} Q}\,[\,M\_par1\,] \qquad \frac{}{\Omega \underset{X}{\parallel} \Omega \xrightarrow{\checkmark} \Omega}\,[\,M\_par2\,]$$

The firing rule for Q is similar to $[M\_par1]$. When both processes become Ω, the parallel process terminates under the firing rule $[M\_par2]$. Notice that the verification results especially on non-terminating checking of parallel composition in CSP$_M$ and CSP# are the same although the former needs two more steps. Parallel processes involving synchronous channels in CSP# have been discussed early in Section 3 (by the firing rule $[\#\_par1]$). Parallel processes involving asynchronous channels execute independently and their firing rules can be found in our technical report [21].

Renaming. CSP$_M$ supports *renaming* which renames a visible event when an associated process is running, shown in the rule $[M\_r3]$. In theory, event renaming $P[[R]]$ can be represented in CSP# by a process Q which is *almost* the same as P except the visible event from relation R being replaced. However, modeling the renaming process manually in CSP# may not be easy when the renaming relation is complicated, and it may lead to larger (LOC) specifications.

$$\frac{P \xrightarrow{\tau} P'}{P[[R]] \xrightarrow{\tau} P'[[R]]}\,[\,M\_r1\,] \qquad \frac{P \xrightarrow{\checkmark} P'}{P[[R]] \xrightarrow{\checkmark} \Omega}\,[\,M\_r2\,] \qquad \frac{P \xrightarrow{a} P', a\ R\ b, a, b \in \Sigma}{P[[R]] \xrightarrow{b} P'[[R]]}\,[\,M\_r3\,]$$

Discussion We have identified differences between CSP$_M$ and CSP# in terms of their operational semantics, and also discussed some possible translations between these two languages, especially their channel communications. Through the analysis, we can draw some general guidelines of their modeling features: CSP$_M$'s adoption of functional paradigm and support of more primitives such as *CHAOS* and *renaming* provide an approach to specify concurrent systems like this, starting with an abstract model first, then refining it to more concrete one. CSP# supports more primitives for modeling different forms of communication (e.g., message passing), and it is feasible to specify concrete system behaviors which require hand shaking, message passing and shared resources. In term of expressiveness, it can be shown that CSP$_M$ and CSP# are equivalent as both CSP$_M$ and CSP# process can be transformed into a normal form, which involves event-prefixing, internal choice and recursion only [15].

4 Verification Tool Support

CSP_M is supported by FDR which is designed primarily for refinement checking in terms of trace, failures, divergences, refusals and revivals. ProB was initially designed as an animator and model checker for B method [1], and recently it supports CSP_M with improvements on static type checking and associative tuples [7]; ProB integrates type checking, animation and model checking together. CSP# is supported by PAT which is an extensible framework for system modeling, simulation and verification. PAT implements a number of model checking techniques catering for different properties such as LTL properties and refinement checking. In the following, Section 4.1 illustrates the verification capabilities of FDR, ProB (for CSP_M) and PAT (only its CSP module), including properties supported and their model checking techniques; Section 4.2 investigates the efficiency of the three tools.

4.1 Verification

FDR, ProB and PAT support the analysis of many common properties such as deadlock, livelock, determinism, and refinement checking which includes trace, failure and failures/divergences refinement. In addition, FDR supports two additional refinement models: the refusal testing model and the revivals model [10]. PAT supports additional properties like *reachability analysis*, i.e., if a system can reach a bad state (e.g., array overflow).

Model checking LTL properties is common in practice. Although it is not directly supported in FDR, the relationship between refinement checking and LTL model checking has been studied (e.g., [16,11]). Particularly, Leuschel et al. [8] applied an emptiness test in a refinement between an unexpected specification and a process; the process is a synchronization of the implementation and a CSP process for an LTL formula. This approach has to deal with the high complexity of synchronization in FDR, and the process to construct CSP processes from LTL formulas is arduous. Lowe [9] used a refusal testing model to conduct the refusal refinement between a CSP process which denotes an LTL formula and its implementation; those supported LTL formulas exclude operators eventually ($\diamond$), until ($\mathcal{U}$), and negation. In contrast, ProB and PAT support various LTL formulas and analysis directly. Moreover, these formulas can constrain both states and events, and be analyzed under five types of fairness assumptions [23] in PAT.

FDR, ProB, and PAT all provide basic model checking techniques such as breadth first search and (bounded) depth first search. In addition, PAT implements the anti-chain approach in which the complete subset construction and computing the complete state space of the product are avoided for checking refinement. Further, PAT applies Loop/SCC searching algorithm for LTL verification under fairness assumptions. To cope with the problem of state space explosion during verification, FDR and PAT develop their own reduction techniques. To be specific, FDR proposes a hierarchical compression approach consisting of six methods to process an LTS representing a CSP_M model [10,15,17]: enumerations, strongly node-labeled bisimulation, τ-loop elimination, diamond elimination, normalization, and factoring by semantic equivalence. On the other hand, PAT deploys three techniques. First, using the atomic sequence construct (denoted by $atomic\{P\}$), where a sequence of statements in a process executes

as one *super-step* without any inference, to realize simple partial order reduction (POR). Second, applying POR dedicated to refinement checking to not only τ transitions but also visible events (in some case which is not supported in FDR [23]). Last but not least, providing process counter abstraction for parameterized systems under fairness against LTL formulas [24]. We remark that the implementation of FDR's hierarchical compression methods for CSP# in PAT is nontrivial due to shared variables supported in CSP#. For instance, a τ event in CSP# may update shared variables and therefore the event cannot not be pruned for compression.

4.2 Experiment

In this section, we evaluate the efficiency of FDR, ProB and PAT by verifying nine benchmark systems. The experiments with FDR and ProB are performed on an Intel® CPU E6550 (2.33 GHz) PC with 4GB memory running on 32-bit Linux. PAT is experimented with the same PC but on a 32-bit Windows.

We conduct five sets of experiments[5]. The first set investigates the performance of refinement checking, by verifying the same model and assertion with different reduction techniques. The results are shown in Table 2, where N is the number of processes. Column $State$ shows the number of visited states, and column $Time(s)$ records running time of the verification in seconds. Value "-" in a cell denotes that the experiment is aborted due to either memory overflow or execution time exceeding two hours. For readers/writers (R/W) models, although FDR applies some dedicated compression techniques, PAT has better performance. For dining philosopher (DP) models, FDR performs extremely well because of the strategy discussed in [18]. However, other experiments show that this strategy may not be as efficient for other models. For Milner's cyclic scheduler (MCS), PAT is comparable to FDR in terms of the number of states per second. FDR processes the LTS by applying its compression methods, whereas PAT applies a simple reduction method, i.e., using the keyword *atomic* to give higher priority to local events which are not synchronized, not updating any variable and not mentioned in the property.

The second set compares the performance of three model checkers on solving puzzles, inspired by work in [12]. The CSP_M and CSP# models for these puzzles make the best use of their modeling power: CSP# specifies the puzzles using shared variables, which are solved by PAT through reachability analysis, whereas CSP_M models the puzzles using multi-part event synchronization, which are solved by FDR and ProB through trace refinement. In addition, FDR simulates a bounded DFS algorithm by searching the divergence of a new system, in order to find a smaller counterexample. The new system P', like a watchdog, can only perform up to N events of the target implementation process P, and then performs an infinite number of events [12]. This approach can be used provided that the target process P is loop-free. Table 3 shows the performance results, where column $FDR\text{-}Div$ records the results of states and time using this algorithm; value $N.A.$ means there is no model with divergence checking to solve the puzzle. From Table 3, we can observe that the divergence checking approach

[5] All models are available at www.comp.nus.edu.sg/~pat/compare.

Table 2. Experiment results on refinement checking

Model	N	Property	FDR		ProB		PAT	
			State	Time(s)	State	Time(s)	State	Time(s)
R/W	6	P [T= S	8	0.024	61365	125.94	9	0.04
R/W	200	P [T= S	202	1.434	-	-	203	0.11
R/W	500	P [T= S	502	19.651	-	-	503	0.057
R/W	1000	P [T= S	1002	156.162	-	-	1003	0.108
DP	6	P [F= S	1	0.06	14510	82.42	1762	0.174
DP	8	P [F= S	1	0.071	-	-	22362	2.995
DP	12	P [F= S	1	0.104	-	-	-	-
MCS	20	P [FD= S	40	0.043	-	-	60	0.114
MCS	50	P [FD= S	100	0.086	-	-	150	0.143
MCS	100	P [FD= S	200	0.246	-	-	300	0.53

Table 3. Experiment results on solving puzzles

Model	N	FDR		FDR-Div		ProB		PAT	
		State	Time(s)	State	Time(s)	State	Time(s)	State	Time(s)
Solitaire	26	4048216	46.303	1	0.169	-	-	11950	5.356
Solitaire	29	28249254	387.737	1	0.217	-	-	104395	54.681
Solitaire	32	-	-	1	5.318	-	-	10955	5.301
Solitaire	35	-	-	1	377.297	-	-	443230	279.454
Knight	5	508450	3.522	1	0.037	-	-	4256	0.29
Knight	6	-	-	1	15.399	-	-	129269	9.143
Knight	7	-	-	1	94.713	-	-	77238	6.754
Hanoi	6	729	0.052	N.A.	N.A.	1667	57.84	5775	0.416
Hanoi	7	2187	0.086	N.A.	N.A.	4969	196.5	92680	6.837
Hanoi	8	6561	0.181	N.A.	N.A.	14853	660.59	150918	11.524

can be used in the solitaire and chess knight tour models. However, this approach cannot always significantly improve the performance, because it depends on the searching order. Moreover, it is costly to check if a system is loop-free or not, which is the premise for applying this approach. PAT solves the two puzzles in a reasonable time, and it is faster in the knight example than FDR and FDR-Div. For the hanoi puzzle, FDR has a better performance because the compression techniques it uses can effectively reduce the state space.

The third set explores the performance of FDR and PAT on verifying two models which involve shared variables. The first example is a concurrent stack which allows multiple readers to access the shared variable at the same time, but only one writer to update the value; readers cannot access the shared variable in the latter case. The modeling of shared variables in CSP$_M$ follows the approach discussed in Section 3. Results of this example in Table 4 show that PAT performs better than FDR for checking trace refinement (P[T=S), and this is because PAT uses DFS with anti-chain algorithm in the trace refinement. This algorithm is effective when the specification is non-deterministic. Here, N is the number of processes and $ConcurrentStack * 2$ in the *Model* column means that the stack size is 2. The second example is the Peterson algorithm. We obtain

Table 4. Experiment results on shared variables

Model	N	Property	FDR		PAT	
			State	Time(s)	State	Time(s)
Concurrent Stack*2	3	P [T= S	453456	3.833	10860	1.023
Concurrent Stack*2	4	P [T= S	-	-	189920	75.915
Concurrent Stack*2	5	P [T= S	-	-	693828	293.382
Peterson	3	mutual exclusion	1011	1.192	3257	0.105
Peterson	4	mutual exclusion	105493	20.067	104686	3.776
Peterson	5	mutual exclusion	14810779	387.645	5722863	294.005

Table 5. Experiment results on LTL checking

Model	N	Property	Result	FDR		ProB		PAT	
				State	Time(s)	State	Time(s)	State	Time(s)
RW	6	$\Box!error$	true	8	0.023	122722	104.8	15	0.059
RW	200	$\Box!error$	true	202	1.455	-	-	403	0.086
RW	500	$\Box!error$	true	502	19.901	-	-	1003	0.071
RW	1000	$\Box!error$	true	1002	154.33	-	-	2003	0.148
DP	6	$\Box\Diamond eat.0$	false	N.A.	N.A.	2420	1.11	166	0.019
DP	8	$\Box\Diamond eat.0$	false	N.A.	N.A.	13312	1.75	256	0.024
DP	12	$\Box\Diamond eat.0$	false	N.A.	N.A.	-	-	460	0.049

the CSP_M model from the shared variable analyzer (SVA) [17]. To be fair, the CSP# model is specified at the same level of granularity as the CSP_M model. The results show that PAT performs better. This is because local events associated as atomic statements in CSP# reduce the states significantly, whereas CSP_M model defines additional events to represent reading/writing operations of shared variables. Although these additional events can be hidden as internal events to apply existing compression techniques in FDR, the effect is minor because the type range of reading/writing channels and operations over different variables can easily lead to state space explosion.

The forth set explores the performance on verifying LTL properties. We adopt the approach proposed by Lowe [9] to construct a CSP_M process for the LTL formula and use FDR to perform the refusal refinement checking. As this approach cannot deal with operator *eventually* ($\Diamond$), we ignore the checking of property $\Box\Diamond eat.0$ in FDR. Table 5 indicates that PAT performs better than FDR and ProB. Notice that property $\Box\Diamond eat.0$ can be verified to be true using PAT under the strong or global fairness assumption.

Last but not least, we conduct a case study on translating CSP_M model to CSP# through a real world problem, namely, the battery monitor component of the elevator control system described in [5]: the CSP_M specification is translated from the battery monitor Simulink diagram, and the CSP# model is translated from the CSP_M model. Then we analyze the Simulink diagram, using the Simulink simulator, to determine which output the battery monitor should produce for every given input. Thus, to assess if both models describe the same behaviour, we compose each one with parallel observers. We noticed that both CSP_M and CSP# models provide the same output value for all relevant scenarios. Although both models were reviewed by CSP_M and CSP# specialists, a formal proof of equivalence will be provided in our future work. Besides this

Table 6. Experiment results on battery monitor

Property	Result	FDR		PAT	
		State	Time(s)	State	Time(s)
Deadlock-free	True	2700	0.286	2700	0.748
Livelock-free	True	2700	0.296	2700	3.723

analysis, we also checked basic properties like deadlock freeness and divergence freeness. The comparison was performed as a controlled experiment and we ran each assertion 30 times. By applying common statistics testing methods (particularly, Shapiro Wilk and Mann-Whitney U [20]) to experiment data, we can state that the difference between PAT and FDR performance is large. From Table 6, we can observe that the visited states in FDR and PAT are the same, and performance of these two tools is similar using the same verification algorithm. Note that the time for deadlock-freeness property in PAT consists of time for deadlock and non-terminating checking. We have not included ProB in this comparison as it is unable to recognize the CSP_M syntax for this example.

Discussion We have explored the supporting tools of CSP_M and CSP#, namely, FDR, ProB and PAT, by comparing their model checking techniques and analyzing their verification capabilities through nine benchmark systems. Our exploration leads to the following four general and practical rules for choosing these tools. First, FDR can be the best candidate when powerful built-in compression techniques are applicable in refinement checking. Second, PAT is a better choice to verify properties of models which involve shared variables. Third, to verify LTL properties, we can use ProB for CSP_M models or FDR for some model where LTL formula can be verified by refusal checking, and PAT for CSP# model. Lastly, PAT may be a better option to handle models where atomic reductions are applicable (e.g., readers/writers and Peterson algorithm).

5 Conclusion

In this work, we presented a comprehensive comparison of CSP_M and CSP#, and their supporting tools FDR, ProB and PAT. We explored their modeling features from the view of their syntax and operational semantics. We also investigated the reasoning power of CSP_M and CSP# in terms of the capability and efficiency of their supporting tools. Our work can guide users to select and assess appropriate modeling languages and reasoning tools for specifying and verifying concurrent systems. 1) CSP_M may be more suitable to model systems with abstract behavior, and systems which involve multi-part event synchronization. On the other hand, CSP# could be a better candidate to handle systems which implement hand shaking or message passing communication mechanisms, and systems which need shared variables. 2) To perform the refinement checking, the decision relies on the reduction techniques which are more applicable (compression methods in FDR, atomic reduction in PAT) to the models. To verify LTL properties, we can use ProB for CSP_M models or FDR for some model (discussed in Section 4), and PAT for CSP# models. Lastly, PAT may be a better option to verify systems with shared variables.

As for related work, Carvalho et al. have made an initial step to explore the differences between CSP_M and CSP# [2]. They compare the two languages from the data and behavioral aspects. Our work here substantially extends their step by considering an in-depth and a wider range of comparisons; for instance, we investigate their intrinsic differences from the operational semantics aspect. Roscoe has briefly described tools which can animate, analyze, and verify CSP models[6]; these tools include FDR, ProB, PAT, ARC [13] and so on. He introduces these tools with strengths and limits from a high level. Our work can be considered as a concrete guideline for these tools, in particular, FDR, ProB for CSP_M, and PAT for CSP#, with intensive experiments.

The comparison of FDR, ProB and PAT so far has been focusing on the classical model checking techniques. In the future, we plan to extend the comparison to other techniques such as SAT-based FDR and BDD-based PAT. Proofs of the semantic equivalence of the translations and implementations of the translators are also our goals.

Acknowledgment. The authors would like to thank Bill Roscoe for the review and suggestions on benchmarks for FDR, Michael Leuschel for the help on using ProB, and Augusto Sampaio, Alexandre Mota and Tarciana Dias for the valuable comments.

References

1. Abrial, J.-R.: The B-book: assigning programs to meanings. Cambridge University Press, New York (1996)
2. Carvalho, G.H.P., Dias, T., Mota, A., Sampaio, A.: Analytical comparison of refinement checkers. In: SBMF, pp. 61–66 (2011)
3. Hoare, C.: Communicating Sequential Processes. Prentice-Hall (1985)
4. Holzmann, G.: Spin model checker, the: primer and reference manual. Addison-Wesley Professional (2003)
5. Jesus, J., Mota, A., Sampaio, A., Grijo, L.: Architectural Verification of Control Systems Using CSP. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 323–339. Springer, Heidelberg (2011)
6. 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)
7. 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)
8. Leuschel, M., Massart, T., Currie, A.: How to Make FDR Spin LTL Model Checking of CSP by Refinement. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 99–118. Springer, Heidelberg (2001)
9. Lowe, G.: Specification of communicating processes: temporal logic versus refusals-based refinement. Form. Asp. Comput. 20(3), 277–294 (2008)
10. Formal Systems (Europe) Ltd.: Failures-Divergence Refinement - FDR2 User Manual (version 2.91)
11. Murray, T.: On the limits of refinement-testing for model-checking CSP. Form. Asp. Comput., 1–38 (2011)
12. Palikareva, H., Ouaknine, J., Roscoe, A.W.: Faster FDR counterexample generation using SAT-solving. ECEASST 23 (2009)

[6] The description is at http://www.cs.ox.ac.uk/ucs/CSPtools.html.

13. Parashkevov, A.N., Yantchev, J.: ARC - a tool for efficient refinement and equivalence checking for CSP. In: ICA3PP, pp. 68–75 (1996)
14. Roscoe, A.W.: CSP is Expressive Enough for Pi (2010)
15. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR (1997)
16. Roscoe, A.W.: On the expressive power of CSP refinement. Form. Asp. Comput. 17, 93–112 (2005)
17. Roscoe, A.W.: Understanding Concurrent Systems. Springer-Verlag New York, Inc. (2010)
18. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M.H., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)
19. Scattergood, B.: The Semantics and Implementation of Machine-Readable CSP. PhD thesis, University of Oxford (1998)
20. Shapiro, S.S., Wilk, M.B.: An analysis of variance test for normality (complete samples). Biometrika 3(52) (1965)
21. Shi, L.: An Analytical and Experimental Comparison of CSP Extensions and Tools. Technical report, NUS (2012), http://www.comp.nus.edu.sg/~pat/compare
22. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: TASE, pp. 127–135 (2009)
23. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
24. Sun, J., Liu, Y., Roychoudhury, A., Liu, S., Dong, J.S.: Fair Model Checking with Process Counter Abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 123–139. Springer, Heidelberg (2009)
25. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.S.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4) (2009)

Symbolic Model-Checking of Stateful Timed CSP Using BDD and Digitization*

Truong Khanh Nguyen[2], Jun Sun[1], Yang Liu[3], and Jin Song Dong[2]

[1] ISTD, Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] School of Computing, National University of Singapore
{truongkhanh,dongjs}@comp.nus.edu.sg
[3] Temasek Lab, National University of Singapore
tslliuya@nus.edu.sg

Abstract. Stateful Timed CSP has been recently proposed to model (and verify) hierarchical real-time systems. It is an expressive modeling language which combines data structure/operations, complicated control flows (modeled using compositional process operators adopted from Timed CSP), and real-time requirements like $deadline$ and $within$. It has been shown that Stateful Timed CSP is equivalent to closed timed automata with silent transitions, which implies that the timing constraints of Stateful Timed CSP can be captured using explicit $tick$ events, through digitization. In order to tackle the state space explosion problem, we develop a BDD-based symbolic model checking approach to verify Stateful Timed CSP models. Due to the rich language features, BDD-based system encoding and verification is highly nontrivial. In this work, we show how to systematically encode Stateful Timed CSP models in BDD. Our approach consists of two steps. The first step is to identify *maximum primitive components* of a given system and then generate finite state machines (FSMs) from them, applying a set of symbolic firing rules. These FSMs are then encoded in the standard way. The second step is to compose the encoded components using a set of BDD-based compositional functions. The proposed method has been implemented in the PAT model checker. It supports properties like reachability, linear temporal logic, etc. The effectiveness of our technique is evaluated with benchmark systems.

1 Introduction

Real-time systems are a class of systems whose correctness depends on the time at which events occur. Examples of real-time systems ranges from simple timed protocols (like Fischer's protocol) to large complex embedded systems (like signaling systems for high-speed trains). The reactions of these systems must obey all of the timing constraints. In other words, these systems must produce responses not only correctly but also with exact timing. Any violation of these constraints may cause damages and risk the human lives. Therefore it is immediately clear that verification real-time systems is a crucial phase in the design of real-time systems.

* This research is partially supported by TRF project 'Research and Development in the Formal Verification of System Design and Implementation'.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 398–413, 2012.

Timed automata are an extension of finite-state automata equipped with finitely many real-valued clock variables to keep track of time. They are often used to model real-time systems. In timed automata, transitions and states may be labeled with clock constraints. Clock constraints labeled with states, called state invariant, limit the amount of time that may be spent at that state. Clock constraints labeled with transitions, called transition guard, must hold for the transition to be taken. To specify properties, a real-time invariant of CTL, e.g., Timed CTL (TCTL, for short) has been proposed.

Efficient automatic model-checking algorithms for real-time systems have been obtained in recent years. Note that traditional model checking algorithms could not be applied directly to real-time verification because time factors are modeled as continuous and real-value variables. Zone abstraction [4], which groups clock valuations using a convex constraint [4], has emerged as a popular approach and has been employed by tools like UPPAAL. Other approaches have also been proposed, especially for a subset of timed automata, which can be digitized (i.e., closed timed automata [5]). For instance, Lamport [10] argued that model checking of real-time systems can be really simple if digitization is adopted. Digitization translates a real-time verification problem to a discrete one by using clock ticks to represent time elapsing explicitly. The advantage of digitization is that the techniques which are developed for classic automata verification can be applied without the added complexity of zone operations. Though digitization does not preserve the continuous-time semantics of time-automata, it was proved to be sound for a large class of verification problems [5].

Stateful Timed CSP has been recently proposed, as a complementary language to timed automata, to model (and verify) hierarchical real-time systems. It is an expressive modeling language which combines data structure/operations, complicated control flows (modeled using compositional process operators adopted from Timed CSP), and real-time requirements like deadline and within. In [21], it has been show that zone abstraction can be applied to Stateful Timed CSP by dynamically creating/deleting clocks. Unsurprisingly, however, state space explosion remains as a huge challenge. In [19], it has been show that Stateful Timed CSP is equivalent to closed timed automata with silent transitions, which implies that the timing constraints of Stateful Timed CSP can be captured using explicit tick events, through digitization. In this work, inspired by on previous work on combining BDD and digitization [3,13], we develop a BDD-based symbolic model checking approach to verify Stateful Timed CSP. Due to the rich language features, BDD-based system encoding and verification is highly nontrivial.

The contribution of the work is threefold. Firstly, we develop a systematic way of encoding Stateful Timed CSP. A Stateful Timed CSP process can be encoded by two ways: using FSMs and using compositional functions. Primitive components are translated to FSMs based on the Stateful Timed CSP semantics. These FSMs are encoded in BDD and then composed gradually by a rich set of compositional functions. Secondly, we support a range of model checking algorithms. For instance, we are able to verify LTL with the assumption of non-Zenoness. While checking whether or not an execution is zeno is difficult for zone approaches [22,6], in digitization, an execution of a digitized system is non-Zeno if and only if it contains infinitely many clock ticks. Therefore a digitized system is non-Zeno if time advances at least one time unit in all its cycles. In other words, non-Zenoness assumption can be supported by requiring all

cycles to contain at least one tick transition. Lastly, we implement our approach in the PAT model checker [20] and evaluate the performance of BDD-based symbolic model checking with zone-based approaches with a number of systems. We show that our approach complements the zone abstraction approach [21] and offers significantly better performance in a number of cases.

Related Work. After the timed automata were introduced in [1], many tools and techniques are proposed, for example, Different Bounded Matrices [4], Clock-Restriction Diagrams [24], and Difference Decision Diagram [12]. Our work was inspired by the digitization which was proposed in [5,10]. However the difference in our symbolic technique is the use of tick transitions to represent explicitly the timing constraints instead of the use of clock variables. Based on this, a BDD encoding library for digitized systems was developed [14]. This paper presents the extension which only focus on verification of Stateful Timed CSP. Our approach is similar to the two-level approach used in FDR [16]. Basically FDR exploits a hybrid high-/low-level approach for calculating the operational semantics of a process. The low level comprises all true recursions while in the high level, processes are composed by parallel composition, hiding and renaming. Identifying low-level processes in FDR is the same as finding the maximum primitive components in our approach. However the ways to tackle the state space explosion in FDR and in our approach are different. In the compiling process on high-level called *super compiling* of FDR, a single LTS is built on-the-fly from other LTSs based on the calculating a set of rules. In contrast in our approach, maximum primitive components are combined by BDD-based compositional functions. Our work in this paper extends the works in [16] because while two-level approach is used to verify un-timed systems, our approach is able to verify real-time systems.

2 Stateful Timed CSP

In this section, we briefly introduce the syntax and the semantics of Stateful Timed CSP processes. The readers are referred to [19] for a complete list of syntax and semantics. Let the label a describe the name of events which are not $tick$ and can be either an external event, a termination event ✓ or an internal event tau, the label c describe channel name and $tick$ denote the passage of one time unit.

A Stateful Timed CSP model is a 3-tuple (Var, σ_0, P_0) where Var is a set of *finite-domain* global variables; σ_0 is the initial valuation of Var (which maps one variable to one value only) and P_0 is a process. A process is a block of computations, which can be defined under Backus-Naur form as Fig. 1.

Process $Stop$ could not make any progress and must still be in the same state after any time period has elapsed. Process $Skip$ is ready to terminate and becomes $Stop$. However some time may elapse before this termination. Process Event Prefixing $a \rightarrow P$ prepares to engage the event a and behaves as P afterward. Similar to $Skip$, delay on this event may occur. Urgent Event Prefixing $a \twoheadrightarrow P$, on the other hand, requires event a to occur as soon as it is enabled. Process Data Operation Prefixing $a\{program\} \rightarrow P$ performs the $program$ with the event a. Note that $program$ can include from simple assignments to complicated sequential structures like if, $while$ and is executed atomically with the

$P = Stop \mid Skip$	– primitives
$\mid a \rightarrow P$	– event prefixing
$\mid a \twoheadrightarrow P$	– urgent event prefixing
$\mid a\{program\} \rightarrow P$	– data operation prefixing
$\mid \text{if}(b)\{P\} \text{ else } \{Q\}$	– conditional choice
$\mid P \mid Q$	– general choice
$\mid P \setminus X$	– hiding
$\mid P;\ Q$	– sequential composition
$\mid P \parallel Q$	– parallel composition
$\mid c?\{program\} \rightarrow P \mid c!\{program\} \rightarrow P$	– Channel Input/Output
$\mid Q$	– process referencing
$\mid Wait[d]$	– delay*
$\mid P\ timeout[d]\ Q$	– timeout*
$\mid P\ interrupt[d]\ Q$	– timed interrupt*
$\mid P\ within[d]$	– timed responsiveness*
$\mid P\ deadline[d]$	– deadline*

Fig. 1. Stateful Timed CSP Process Constructs

event. Process Conditional Choice, defined as if$(b)\{P\}$ else $\{Q\}$ will behave as P or as Q based on the evaluation of the expression b. Process Unconditional Choice $P \mid Q$ offers an (unconditional) choice between P and Q[1]. Sequential composition $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. Process $P \setminus X$ hides occurrences of events in X from the environment. In other words, any event in X engaged by P becomes invisible event τ. Parallel composition of two processes P and Q is written as $P \parallel Q$, where P and Q may communicate via event synchronization (following CSP rules [7]) or shared variables. Notice that if P and Q do not communicate through event synchronization, then it is written as $P|||Q$, which reads as 'P interleave Q'. In addition to multi-party synchronization based on event names, Stateful Timed CSP also provides pairwise synchronization via channel communications. Transitions labeled with channel input (or channel output) of a process can not be taken on its own but must be matched by transitions labeled with corresponding channel output (channel input) of another process running in parallel with it. A process may be given a name, written as $P \mathrel{\widehat{=}} Q$, and then referenced through its name. Recursion is allowed by process referencing.

In addition to two traditional timed process constructs Delay ($Wait$), and Timeout ($timeout$) from Timed CSP, Stateful Timed CSP includes three new process constructs Time Interrupt ($interrupt$), Timed Responsiveness ($within$) and Deadline ($deadline$). This extension allows us to capture common real-time system behavior patterns easily (all timed process constructs are marked with * in Figure 1). Let $d \in \mathbb{R}_+$. Process $Wait[d]$ idles for exactly d time units before terminating. Process $P\ timeout[d]\ Q$ imposes a constraint on the process P to engage the first visible event within d time units. Otherwise after d time units, process Q takes the execution control. In process $P\ interrupt[d]\ Q$, if P terminates before d time units, $P\ interrupt[d]\ Q$ behaves exactly as P. Otherwise, $P\ interrupt[d]\ Q$ behaves as P until d time units and then Q

[1] For simplicity, we omit external and internal choices [7] in the discussion.

takes over. In contrast to $P\ timeout[d]\ Q$, P may engage in multiple visible events before it is *interrupted*. Process $P\ within[d]$ requires process P to engage an visible event with in d time units. In process $P\ deadline[d]$, P must terminate within d time units, possibly after engaging in multiple visible events. Notice that a timed process construct is always associated with an *integer* constant d which is referred to as its parameter.

Example 1. We use Fischer's mutual exclusion protocol [9] to illustrate system modeling using Stateful Timed CSP. The protocol is designed to guarantee mutually exclusive access to a critical section among competing processes $P(i)$ where $i \in [1..n]$ is the unique identifier of that process. Each process $P(i)$ executes the following algorithm where $lock$ is a shared variable, and initialized with the value 0:

repeat
  await$(lock = 0)$;
  $lock := i$
  delay
until $(lock = i)$;
$critical\ section$;
$lock := 0$;

Note that **await** $(cond)$ is an abbreviation for **while** $(\neg\ cond)$ **do** $skip$ and **delay** corresponds to an explicit delay statement. The role of the **delay** statement is that it guarantees while it delays itself, other processes after passing the **await** statement must finish the assignment $lock := i$. The correctness of the protocol depends on the assumptions about the time taken to read and write to the shared variable $lock$, and the delay length. It was shown that the mutual exclusion is guaranteed if the upper bound a on the time taken at the assignment $lock := i$ is less than the lower bound b on the delay length. Because other reading and writing statements to the shared variable $lock$ is not important, we will not impose any timing constraint on them. The protocol can be modeled as a Stateful Timed CSP model $(Var, \sigma_0, Fischer)$ where $Var = \{lock\}$ and $\sigma_0(lock) = 0$ and process $Fischer$ is defined as: $P(1)|||\cdots|||P(n)$ where

$$
\begin{aligned}
P(i) \quad \widehat{=}\ & if(lock = 0)\{ \\
& \quad (setLock\{lock := i\} \rightarrow Skip)\ deadline[a]; \\
& \quad Wait[b]; \\
& \quad if(lock = i)\{ \\
& \quad\quad Critical(i) \\
& \quad \}else\{ \\
& \quad\quad P(i) \\
& \quad \} \\
& \}; \\
Critical(i) \widehat{=}\ & enter \rightarrow exit\{lock := 0\} \rightarrow P(i);
\end{aligned}
$$

Process $Fischer$ is the Interleave composition of $P(1)|||\cdots|||P(n)$. Each process $P(i)$ has an unique identifier described as i. As we can see in the model, timing constraints on each operation can be translated straightforwardly using the set of timed process constructs. For example, $(setLock\{lock := i\} \rightarrow Skip)\ deadline[a]$ imposes a constraint

on the event $setLock$, i.e., it must occur within a time units. The **delay** statement which delays at least b time units can be expressed as $Wait[b]$. Note that after waiting exactly b time units in $Wait[b]$, the process $P(i)$ behaves as the process if$(lock = i)\{\cdots\}$. Since we do not put any constraint on this process, it can idle as long as it wants. Therefore in total the process $P(i)$ can delay at least b time units before entering the critical section. □

There are two approaches to verify Stateful Timed CSP. One is based zone abstraction, which has been proposed in [19]. The other is through digitization, since it has been proved that Stateful Timed CSP is equivalent to some variant of closed timed automata [19]. On one hand, while zone abstract works well in many examples, its complexity is exponential in the number of clocks and its performance in practice can be strongly related to ratio of constants appearing in the clock constraints. For instance, in the Leader Algorithm (which has a very small maximal constants of clock constraints), Uppaal's execution time is strongly dependent on the ratio $MsgDelay/Period$ [10]. Specifically for ratios greater than 0.6, Uppaal easily runs out of memory. On the other hand, though digitization suffers from large clock upper bounds (which imply a large number of tick events), it is not affected by the ratio of the constants. Furthermore, some problems like the non-Zenoness checking problem are much easier with digitization. We thus proposed an approach complementary to the zone abstraction approach in [19], using BDD and digitization to verify Stateful Timed CSP.

3 BDD Encoding

In this section, we show how we systematically encode Stateful Timed CSP processes in BDD. There are two ways. One is to generate an FSM for each Stateful Timed CSP process and encode the FSM in the standard way. The other is to define a set of BDD compositional functions according to the process construct semantics and then encode Stateful Timed CSP processes into BDDs directly without the FSM construction. Both have their own advantages and therefore are used in different cases.

We remark that Stateful Timed CSP is expressive enough so that a process expression generated by the operational semantics may be unbounded. For example, define $P_0 = e \rightarrow (P_0 ||| P_{new})$ which forks a process P_{new} every time e occurs. The resultant process therefore may contain unboundedly many copies of P_{new}. In this work, we assume that a process always has a bounded length, following [17,15].

3.1 Encoding Stateful Timed CSP Processes with FSMs

An FSM is a tuple $\mathcal{M} = (Var, S, init, Act, T)$ such that Var is a set of finite-domain variables; S is a finite set of control states; $init \in S$ is the initial state; Act is the alphabet of events and channels; and T is a labeled transition relation. A transition label is of the form $[guard]evt\{prog\}$ where $guard$ is an optional guard condition constituted by variables in Var; evt is either an event name, a channel input/output or the special $tick$ event (which denotes 1-unit elapsed time); and $prog$ is an optional transaction, i.e., a sequential program which updates global/local variables. A transaction (which may

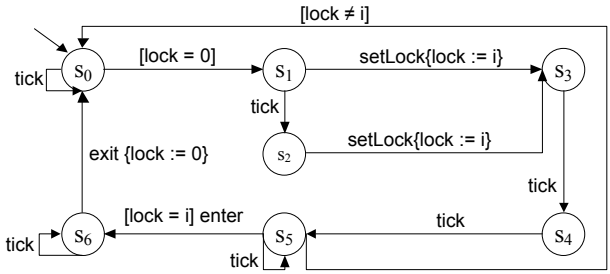


Fig. 2. The FSM of Process $P(i)$

contain program constructs like $while\text{-}do$) associated with a transition is to be executed atomically. A non-atomic operation can be broken into multiple transitions. A transition is possible if the $guard$ is true given current valuation σ of Var. Moreover a transition labeled with channel input/output can not occur by itself but must be synchronized with the transition labeled with corresponding channel output/input.

The operational semantics of Stateful Time CSP allows us to interpret Stateful Time CSP processes as FSMs. For example, we can manually draw the FSM shown in Fig. 2 for the process $P(i)$ of Fischer's protocol in the Example 1 with $a = 1$ and $b = 2$. However translating from a Stateful Time CSP process to an FSM in general is not trivial. In this following, we show how to systematically build the corresponding FSM from a Stateful Timed CSP process. This approach relies on symbolic firing rules, which are different from concrete firing rules in [21] as variables valuations are irrelevant. Specifically the symbolic firing rules are used to generate the whole control flow of a certain process. In other words, the valuation of variables and the effect of transactions are ignored at this step, but they will be considered when transactions are encoded in BDD. For instance, the symbolic firing rule of process Data Operation Prefixing $Q = b\{x := x + 1\} \rightarrow R$ says that at the process Q, if the transition labeled with $b\{x := x + 1\}$ is taken, it will behave as R. In contrast concrete firing rules say, e.g., at the process Q, suppose the current value of x is 0, then after the transition is taken, it will be have as R and the value of x becomes 1. The concrete firing rules, therefore, are used to generate on-the-fly the whole state space explicitly. So different uses of firing rule are suitable for different purposes. In this work, symbolic firing rules are used to generate the corresponding FSM systematically and effectively. Our symbolic firing rules follow the form in [18]:

$$\frac{\begin{array}{l} antecedent\ 1 \\ \cdots \\ antecedent\ n \end{array}}{conclusion} \quad [\ side\ condition\ ]$$

The conclusion can be deduced if all the antecedents are true and the side condition is also true. In the case where antecedents or side condition are missing, they are considered

$$\frac{}{(a \to P) \xrightarrow{a} P} \qquad \frac{}{(a \to P) \xrightarrow{tick} (a \to P)}$$

$$\frac{P_0 \xrightarrow{[g]a\{p\}} P_0'}{\begin{array}{l} P_0|||P_1 \xrightarrow{[g]a\{p\}} P_0'|||P_1 \\ P_1|||P_0 \xrightarrow{[g]a\{p\}} P_1|||P_0' \end{array}} \; [\, a \neq \checkmark \,] \qquad \frac{\begin{array}{l} P_0 \xrightarrow{tick} P_0' \\ P_1 \xrightarrow{tick} P_1' \end{array}}{P_0|||P_1 \xrightarrow{tick} P_0'|||P_1'}$$

$$\frac{\begin{array}{l} P_0 \xrightarrow{[g_0]\checkmark\{p_0\}} P_0' \\ P_1 \xrightarrow{[g_1]\checkmark\{p_1\}} P_1' \end{array}}{P_0|||P_1 \xrightarrow{[g_0 \wedge g_1]\checkmark\{p_0;\, p_1\}} P_0'|||P_1'} \qquad \frac{\begin{array}{l} P_0 \xrightarrow{[g_0]c?\{p_0\}} P_0' \\ P_1 \xrightarrow{[g_1]c!\{p_1\}} P_1' \end{array}}{\begin{array}{l} P_0|||P_1 \xrightarrow{[g_0 \wedge g_1]c\{p_0;\, p_1\}} P_0'|||P_1' \\ P_1|||P_0 \xrightarrow{[g_0 \wedge g_1]c\{p_0;\, p_1\}} P_1'|||P_0' \end{array}}$$

$$\frac{}{Wait[t] \xrightarrow{tick} Wait[t-1]} \; [\, t \geq 1 \,] \qquad \frac{}{Wait[0] \xrightarrow{\tau} SKIP}$$

$$\frac{P_0 \xrightarrow{a} P_0'}{P_0 \; within[t] \xrightarrow{a} P_0'} \qquad \frac{P_0 \xrightarrow{\tau} P_0'}{P_0 \; within[t] \xrightarrow{\tau} P_0' \; within[t]}$$

$$\frac{P_0 \xrightarrow{tick} P_0'}{P_0 \; within[t] \xrightarrow{tick} P_0' \; within[t-1]} \; [\, t \geq 1 \,]$$

Fig. 3. Sample Symbolic Firing Rules

as vacuously true. A number of conclusions which can be drawn from the same set of antecedents and side condition can be grouped below the line one after the other.

The FSM generation procedure basically works as follow. Each process P is mapped with a state in the FSM called 'state P' and this state is also the initial state of that process's FSM. There is a transition labeled with $[guard]evt\{prog\}$ from state P to state P' when the relation $P \xrightarrow{[guard]evt\{prog\}} P'$ can be deduced from the rules. The symbolic firing rules are applied until there is no new state generated. In the following, we present the sample symbolic firing rules of Event Prefixing, Interleave, Delay, and Timed Responsiveness process constructs.

– Given any process Event Prefixing $a \to P$, there is a transition labeled with event a from the state $a \to P$ to the state P. In addition, there is a transition label with event $tick$ looping at the state $a \to P$. For the events marked as urgent, this looping transition labeled with event $tick$ is not available. It forces the process to engage the event without any delay.

- Based on rules of process Interleave, all of the subprocesses in the Interleave composition must synchronize with the termination ✓ and $tick$ events. Moreover channel in transition labeled with $c?$ from one process can be combined with channel out transition labeled with $c!$ from another process to be promoted as c. When transitions are synchronized, we constraint transactions of these transition are not conflict and the execution order of transactions are not important. Other events occur interleave. In addition in the symbolic firing rules of this process and also of other processes, $tick$ transitions are never attached with any guard condition and any transaction. They are simple as a direct sequence of the use $tick$ transitions to explicitly represent the timing constraints. This simplicity helps us to have more optimal BDD encoding of the $tick$ transitions.
- $Tick$ transitions are used to track the passage of one time unit in the symbolic firing rules of process Delay $Wait[t]$. Specifically there is a transition labeled with $tick$ from the state $Wait[t]$ to state $Wait[t-1]$. After delaying itself, it will behave as $SKIP$ by the τ transition from state $Wait[0]$ to state $SKIP$.
- The last three rules are the symbolic firing rules of process construct Timed Responsiveness P_0 $within[t]$. These rules are self-explanatory. $Tick$ transitions are used to track the passage of time. Unless a visible event is engaged, the timed responsiveness condition is not resolved.

Example 2. Process $P(i)$ of Fischer's protocol in the Example 1 is used again as illustration. However for simplicity all the states are renamed to $s_0, \cdots, s_6$ and we will explain the FSM generation procedure starting at process $Critical(i)$ whose corresponding state is the state s_5. According to the firing rules of process Event Prefixing in Fig. 3, in the FSM of the process $P(i)$, there is a transition labeled with $[lock = i]enter$ from the state $Critical(i)$ (state s_5) to state $exit\{lock := i\} \rightarrow P(i)$ (state s_6), and a transition labeled with $tick$ looping at the state of process $Critical(i)$ (state s_5). Then by applying those firing rules again for the process $exit\{lock := i\} \rightarrow P(i)$, there is another transition labeled with $exit\{lock := i\}$ from the state $exit\{lock := i\} \rightarrow P(i)$ (state s_6) back to the state $P(i)$ (state s_0), and a transition labeled with $tick$ looping at the state $exit\{lock := i\} \rightarrow P(i)$ (state s_6). The FSM generation procedure is stopped because there is no new state created. □

Before giving explanation how to encode an FSM, we will briefly describe how to encode a finite set. Essentially given any finite set X, encoding X is to enumerate elements of X in binary and represent them as Boolean functions. Therefore to encode X, we need n boolean variables $x_0, \cdots, x_{n-1}$ where $n = \lceil log_2 |X| \rceil$. Then each element in X is mapped with a bit vector $(x_0, \cdots, x_{n-1})$ by an injective encoding function $f_X : X \rightarrow \{0,1\}^n$. Note that this mapping is fixed throughout the BDD encoding. For instance, encoding the set of four elements $X = \{a, b, c, d\}$ requires two boolean variables x_0 and x_1. The encoding functions f_X is defined as $f_X(a) = (0,0)$, $f_X(b) = (0,1)$, $f_X(c) = (1,0)$, and $f_X(d) = (1,1)$. As a result the predicate of the subset $Y = \{a, b\}$ is $((x_0, x_1) = f_X(a) \vee (x_0, x_1) = f_X(b))$. For simplicity we will use the label x to denote the bit vector $(x_0, \cdots, x_{n-1})$. Therefore the predicate of the subset Y can be rewritten shortly as $(x = f_X(a) \vee x = f_X(b))$. Using this technique, we can encode the set of states and the set of event names and channel names in an FSM. Moreover we

can also encode all the data types whose domain is finite, e.g., boolean, integer, array of booleans, and array of integers. To encode transitions, each variable x in $\overrightarrow{V} \cup \overrightarrow{v}$ has another copy called x' which denotes the variable x's value after the transition.

The BDD encoding of an FSM, referred to as a *BDD machine*, is a tuple $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$. $\overrightarrow{V}$ is a set of unprimed Boolean variables encoding global variables, event names and channel names, which are fixed for the whole system before encoding. $\overrightarrow{v}$ is a set of variables encoding local variables and local control states; $Init$ is a formula over $\overrightarrow{V}$ and $\overrightarrow{v}$ encoding the initial valuation of the variables. $Trans$ is the encoding of transitions *excluding synchronous channel input/output and tick-transitions*. Out (In) is the encoding of synchronous channel output (input). Note that transitions in Out and In are to be matched by corresponding transitions in In and Out respectively from the environment and are thus separated from the rest of the transitions. $Tick$ is also the encoding of transitions labeled with $tick$. Then the final transition function of an FSM is taken from $Trans$ and $Tick$. In other words, it can engage an action or idle one time unit. We still calculate Out and In and separate them from $Trans$ and $Tick$ because transitions from Out and In can be useful if they are synchronized.

Let *BDD machine* $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ be the encoding of an FSM $\mathcal{M} = (Var, S, init, Act, T)$ where

- $\overrightarrow{V} = V_1 \cup Events$ where V_1 and $Events = \{event_0, \cdots, event_{n-1}\}$ are the sets of boolean variables to encode global variables and the alphabet Act respectively. Let $event$ denote the bit vector $(event_0, \cdots, event_{n-1})$.
- $\overrightarrow{v} = v_1 \cup States$ where v_1 and $States = \{state_0, \cdots, state_{m-1}\}$ are the sets of boolean variables to encode local variables and the set of states S respectively. Similarly let $state$ denote the bit vector $(state_0, \cdots, state_{m-1})$. Moreover for any global or local variable x, let the same label x denote the corresponding bit vector of boolean variables to encode that variable. Note that these labels x are different. The former x is the variable declared in the model while the latter x is a shorthand for a bit vector in the BDD encoding functions.
- $Init = (state = f_S(init))$
- $Trans = \bigvee(state = f_S(s_0) \wedge g_{bdd} \wedge event' = f_{Act}(e) \wedge prog_{bdd} \wedge state' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with $[g]e\{prog\}$ (where $e \neq tick$). For simplicity, we skip how we encode guard expression g to g_{bdd} and program block $prog$ to $prog_{bdd}$. Interested readers can refer to [13].
- $Out = \bigvee(state = f_S(s_0) \wedge g_{bdd} \wedge event' = f_{Act}(e) \wedge prog_{bdd} \wedge state' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with a synchronous channel output e, guarded with g and attached with transaction $prog$.
- $In = \bigvee(state = f_S(s_0) \wedge g_{bdd} \wedge event' = f_{Act}(e) \wedge prog_{bdd} \wedge state' = f_S(s_1))$ for all transitions from state s_0 to state s_1 labeled with a synchronous channel input e, guarded with g and attached with transaction $prog$.
- $Tick = \bigvee(state = f_S(s_0) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_1))$ for all tick transitions from state s_0 to state s_1.

Example 3. The *BDD machine* $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ of the FSM in Fig. 2 is as follow:

- $\overrightarrow{V} = Lock \cup \{event_0, event_1\}$ where $Lock$ is the set of boolean variables to encode the shared variable $lock$.
- $\overrightarrow{v} = \{state_0, state_1, state_2\}$. Note that the process parameter i in the definition of $P(i)$ is constant and is replaced with its value before the encoding. In the below encoding functions of $Trans$ and $Tick$, we still keep i to show generally how all processes $P(i)$ in the Fischers' protocol are encoded.
- $Init = (state = f_S(s_0))$
- $Trans = (state = f_S(s_0) \wedge lock = 0 \wedge state' = f_S(s_1))$
  $\vee\ (state = f_S(s_1) \wedge event' = f_{Act}(setLock) \wedge lock' = i \wedge state' = f_S(s_3))$
  $\vee\ (state = f_S(s_2) \wedge event' = f_{Act}(setLock) \wedge lock' = i \wedge state' = f_S(s_3))$
  $\vee\ (state = f_S(s_5) \wedge lock \neq i \wedge state' = f_S(s_0))$
  $\vee\ (state = f_S(s_5) \wedge lock = i \wedge event' = f_{Act}(enter) \wedge state' = f_S(s_6))$
  $\vee\ (state = f_S(s_6) \wedge event' = f_{Act}(exit) \wedge lock' = 0 \wedge state' = f_S(s_0))$
- $Out = In = false$
- $Tick = (state = f_S(s_0) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_0))$
  $\vee\ (state = f_S(s_1) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_2))$
  $\vee\ (state = f_S(s_3) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_4))$
  $\vee\ (state = f_S(s_4) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_5))$
  $\vee\ (state = f_S(s_5) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_5))$
  $\vee\ (state = f_S(s_6) \wedge event' = f_{Act}(tick) \wedge state' = f_S(s_6))$ □

3.2 Encoding Stateful Timed CSP Processes with Compositional Functions

By using the approach presented in the last section, in theory we can translate any Stateful Timed CSP process to an FSM and encoding it. However we do not apply that approach to generate the FSM of parallel processes. Because in the FSM of a parallel composition, the numbers of states and transitions grow exponentially with the number of subprocesses running in parallel. Especially it becomes completely redundant when guards and transactions of the transitions in a certain sub-process are encoded to BDD many times. For example, if we apply the FSM generation procedure to the process $P_1 ||| P_2$, suppose the state of that FSM is of the form (s_1, s_2) where s_1, and s_2 are states in the FSMs of P_1 and P_2 respectively. For any transition t from state s_1 to s_1' in the FSM of P_1, there is a corresponding transition from state (s_1, s_2) to state (s_1', s_2) in the FSM of $P_1 ||| P_2$. Obviously the guard and the transaction of the transition t will be encoded m times where m is the number of states in the FSM of P_2. These overheads make encoding of parallel processes with FSMs inefficient. Therefore we provide compositional functions to encode parallel processes without translating it to FSMs. As a result, compositional functions for all kinds of processes are required to be provided because after using the compositional function, the FSM is no longer available and only compositional functions can be used.

In the following, we will show how to encode two kinds of Stateful Timed CSP processes: Interleave and Timed Responsiveness processes with compositional functions. We fix two BDD machines $\mathcal{B}_i = (\overrightarrow{V}, \overrightarrow{v}_i, Init_i, Trans_i, Out_i, In_i, Tick_i)$, $i \in \{0, 1\}$, which are the encoding of processes P_i. $\overrightarrow{v}_0$ and $\overrightarrow{v}_1$ are disjoint and $\overrightarrow{V}$ is always shared. Symbolic firing rules of Interleave and Timed Responsiveness process

constructs in Fig. 3 can be refered to follow the compositional encoding. Interested readers can refer to [13] for the complete list.

Interleave: Process Interleave can contains 2 or more subprocesses running in parallel. Different from process Parallel these processes are only synchronized in termination event ✓ (still has pairwise synchronization in channel communication like Parallel). Let $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ be the BDD machine encoding of the Interleave composition of two processes P_0 and P_1 such that:

- $\overrightarrow{v} = \overrightarrow{v}_0 \cup \overrightarrow{v}_1$;
- $Init = Init_0 \wedge Init_1$.
- $Trans = \bigvee_{i\in\{0,1\}}[(Trans_i \wedge event' \neq f_{Act}(\checkmark) \wedge \overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i}) \vee (In_i \wedge Out_{1-i}) \vee (Trans_i \wedge Trans_{1-i} \wedge event' = f_{Act}(\checkmark))]$. $Trans$ includes 3 kinds of transitions: local transitions from each component, synchronous channel communication and synchronous termination transition. $(\overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$ denotes that the local variables of $\mathcal{B}_{1-i}$ are unchanged.
- $In = \bigvee_{i\in\{0,1\}}(In_i \wedge \overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$
- $Out = \bigvee_{i\in\{0,1\}}(Out_i \wedge \overrightarrow{v}_{1-i} = \overrightarrow{v}'_{1-i})$
- $Tick = Tick_0 \wedge Tick_1$

Within: In P_0 $within[t]$ process, process P_0 is forced to engage a visible event within the given t time units. Let $\mathcal{B} = (\overrightarrow{V}, \overrightarrow{v}, Init, Trans, Out, In, Tick)$ be the BDD machine encoding of P_0 $within[t]$ where

- $\overrightarrow{v} = \overrightarrow{v}_0 \cup \{clk\}$, $-1 \leq clk \leq t$ records the number of elapsed time units so far and $clk = -1$ indicates an visible action is engaged.
- $Init = (Init_0 \wedge clk = 0)$
- $Trans = clk \leq t \wedge Trans_0 \wedge [(event \neq f_{Act}(\tau) \wedge clk' = -1) \vee (event' = f_{Act}(\tau) \wedge clk' = clk)]$
- $In = clk < t \wedge In_0 \wedge clk' = -1$
- $Out = clk < t \wedge Out_0 \wedge clk' = -1$
- $Tick = Tick_0 \wedge [(clk \geq 0 \wedge clk < t \wedge clk' = clk + 1) \vee (clk = -1 \wedge clk' = -1)]$

Note that a channel communication is clearly a visible event. Thus if channel communication occurs, variable clk is assigned -1 to mark the happening of that visible event.

As we can observe, except parallel processes, encoding processes with compositional functions is not as optimal as with FSMs. Unlike encoding with FSMs, many auxiliary variables are introduced in the encoding with compositional functions to control the flow, for example, clk variable in Timed Responsiveness to record the number of elapsed time units. Therefore our strategy for encoding a Stateful Timed CSP process is to find its maximum primitive components which can be translated to FSMs and then encode these FSMs as BDD machines. Identifying the maximum primitive components is straightforward because maximum primitive components are the maximum components whose definitions do not contain Parallel/Interleave process construct. Finally these BDD machines are composed to achieve the final BDD machine of the given process. For instance, in Example 1, the identified maximum primitive components are

n processes $P(i)$ where $i \in \{1, \cdots, n\}$. Next, FSMs translated from these subprocesses are encoded as BDD machines, which are then composed using the Interleave compositional function to generate the BDD encoding of the process $Fischer$.

3.3 Limitations on BDD Encoding

Stateful Timed CSP is too expressive to be fully encoded. Consequently there are some Stateful Timed CSP processes which are not possible to be encoded. Firstly processes having varying parameters are not supported. An example of processes having a varying parameter is $P(i) = a \rightarrow P(i+1)$. The reason of this limitation is because of the update of the parameter $i := i + 1$ when the process starts to behave as P again. This update must be done somewhere before the process behaves as P again. There are two possible ways to deal with this, one is to attach the parameter updates on the immediately precedent event (in this example it is the event a), another is to create a separate transition to update the process parameters. However both ways have problems which may change the semantics of the defined process. In the first way, these parameter updates could conflict with each other at the precedent event. An illustration of this problem is $Q = a \rightarrow (P(1) \mid P(2))$ where after event a, there is a choice between $P(1)$ and $P(2)$. Therefore we have two conflict updates of the process parameter i of process P, $i := 1$ and $i := 2$. In the second way, by introducing new transitions which updates process parameters, there is a question on the semantics of these transitions, specifically whether these transitions can resolve the choice. If these transitions do not resolve the choice, in the last example, two transitions which update $i := 1$ and $i := 2$ respectively can happen before the choice is resolved. This is similar to the problem in the first way where there are conflicts between these parameter updates. On the other hand, if these transitions can resolve the choice, suppose that in the last example, $P(1)$ could not engage any event while $P(2)$ can, then the process can take the transition which updates $i := 1$ and resolve the choice in favor of the process $P(1)$. After that the process becomes deadlock. However this could not happen because since $P(1)$ is deadlock, the choice must be resolved in favor of $P(2)$.

Secondly encoding with compositional functions could not be applied to recursive processes, e.g., $P = a \rightarrow P$. Based on the Stateful Timed CSP semantics, encoding with compositional function is used to achieve the encoding of a process based on the known encodings of subprocesses. Therefore it is obvious that using compositional functions on a process whose definition has a reference call to itself is not possible and will create an infinite recursive calls of the compositional functions.

In summary there are two restrictions on BDD encoding of Stateful Timed CSP. One restriction is that processes must have constant parameters. However there is a small number of models requiring varying parameters. Moreover global variables can be used to alleviate the restriction. By promoting each varying process parameter with a corresponding global variable and manually attaching the update of those global variables to the suitable events, an equivalent model can be achieved. The other restriction is that compositional encoding is not available for recursive processes and yet this restriction is inevitable. Remember that introducing compositional functions is to optimize the encoding of the parallel process which is the main cause of the state space explosion. After the use of compositional functions, only encoding by compositional functions is

possible. However in our experience often the recursive processes do not contain parallel composition. Consequently these processes can be encoded using FSMs.

4 Implementation and Evaluation

Our technique has been implemented as part of the PAT framework [20]. It is based on the CUDD package, with about thirty classes and thousands of lines of C# code. The implementation includes two parts: encoding and verification. The encoding part has functions to generates the FSM from Stateful Timed CSP processes. The advantage of our technique is that the FSM generation procedure is very simple, yet systematical and efficient. For each process construct we only need to define what transitions can be taken from that process and then these transitions are added from the state of the current process. This procedure is called recursively in subsequent processes. In addition to the FSM generation procedure, the encoding part also contains a function to encode an FSM and a set of compositional functions for all process constructs. The second part is the verification which supports a range of properties, e.g., reachability and deadlock analysis or LTL. Verification of LTL is based on a symbolic implementation of the automata-based approach [8,23]. By using digitization technique, verification of real time system, specifically Stateful Timed CSP becomes feasible. Digitization translates a real-time verification problem to a discrete one by using clock ticks to represent elapsed time. Therefore the current model checking algorithm for concurrent systems can be applied without the added complexity of zone operations. Moreover verification of LTL with non-Zenoness assumption can also be supported by converting the non-Zenoness assumptions as *justices* conditions (weak fairness) [8]. In the following, we evaluate our technique in verification Stateful Timed CSP by comparing its performance with the zone-based approach in PAT in many examples. All models are available online [13]. The test bed is a PC with Intel Core 2 Duo E6550 CPU at 2.33GHz and 3GB RAM.

According to the experiment results in Table 1, in the verification of three mutual exclusion protocols Fischer's protocol [9], AT92 [2], and LTS92 [11], BDD-based approach consistently outperforms Zone-based approach. BDD-based approach is not only faster but also uses less memory than Zone-based approach. For instance, in Fischer protocol of 6 processes, zone-based approach takes more than 1000 seconds and 215 MBs while BDD-based takes only 6 seconds and 101 MBs. Moreover zone-based approach runs out of memory with Fischer protocol of 7 processes, yet BDD-based approach can verify the protocol of up to 12 processes. However in the verification of Train Controller, zone-based approach is much better than BDD-based approach. For instance, in the Railway Controller with 6 trains, zone-based approach only takes 4 seconds and 18 MBs but BDD-based approach takes 905 seconds and 1458 MBs. The reason for this considerable change in performance of BDD-based approach is because of two issues. First the size of BDDs is very sensitive to large clock values. In this benchmark, we set the maximal clock constant to small values, e.g., 4 for Fischer's protocol and 3 for others. Second the size of BDDs is also very sensitive with the FSMs of processes. After examining many examples, we find that there are some models where it is difficult to fully take advantage of the data-sharing capability of BDDs. This is the reason why although we have reduced the maximal clock constants to a very small

Table 1. Compare Zone-based Approach and BDD-based Approach

Model	#Processes	Zone		BDD	
		Time (s)	Memory (MB)	Time (s)	Memory (MB)
Fischer	5	44	19	2	43
Fischer	6	1283	215	6	101
Fischer	7	x	x	17	231
Fischer	12	x	x	1112	1353
AT92	3	7	26	1	22
AT92	4	770	524	2	36
AT92	5	x	x	14	163
AT92	8	x	x	2880	1684
LS92	4	2	13	1	24
LS92	5	1292	76	1	35
LS92	6	x	x	3	57
LS92	15	x	x	996	1406
Railway Controller	5	1	10	51	650
Railway Controller	6	4	18	905	1458
Railway Controller	7	24	18	x	x
Railway Controller	8	201	557	x	x

value, the BDD-based approach's performance is still much poorer than zone-based approach in the Railway Controller example. In contrast if data-sharing occurs a lot in BDDs, the efficiencies of BDD would be higher. This can be shown when we increase the maximal clock constants of the first three protocols up to 20, BDD-based approach still outperforms zone-based approach. Specifically BDD-based approach can verify Fischer's protocol of 10 processes, AT92 of 5 processes and LTS92 of 8 processes. In summary there are some models where zone-based approach performs well while there are other models where BDD-based approach performs well. This experiment shows that these two approaches complements each other.

5 Conclusion

We have illustrated our approach to verify Stateful Timed CSP by using BDD and digitization. We have also presented how Stateful Timed CSP processes are systematically encoded with FSMs and compositional functions. Furthermore our experiments show that there is no superior approach but these approaches have different but complementary advantage.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science 126, 183–235 (1994)
2. Alur, R., Taubenfeld, G.: Results about Fast Mutual Exclusion. In: IEEE Real-Time Systems Symposium, pp. 12–22 (1992)

3. Beyer, D., Lewerentz, C., Noack, A.: Rabbit: A Tool for BDD-Based Verification of Real-Time Systems. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 122–125. Springer, Heidelberg (2003)
4. Dill, D.L.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
5. Henzinger, T.A., Manna, Z., Pnueli, A.: What Good Are Digital Clocks? In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 545–558. Springer, Heidelberg (1992)
6. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Efficient Emptiness Check for Timed Büchi Automata. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 148–161. Springer, Heidelberg (2010)
7. Hoare, C.A.R.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall (1985)
8. Kesten, Y., Pnueli, A., Raviv, L.-O.: Algorithmic Verification of Linear Temporal Logic Specifications. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 1–16. Springer, Heidelberg (1998)
9. Lamport, L.: A Fast Mutual Exclusion Algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
10. Lamport, L.: Real-Time Model Checking Is Really Simple. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 162–175. Springer, Heidelberg (2005)
11. Lynch, N.A., Shavit, N.: Timing-Based Mutual Exclusion. In: IEEE Real-Time Systems Symposium, pp. 2–11 (1992)
12. Møller, J.B., Hulgaard, H., Andersen, H.R.: Symbolic Model Checking of Timed Guarded Commands Using Difference Decision Diagrams. J. Log. Algebr. Program. 52-53, 53–77 (2002)
13. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S., Liu, Y.: BDD-based Discrete Analysis of Timed Systems (2012), http://www.comp.nus.edu.sg/%7Epat/bddlib
14. Nguyen, T.K., Sun, J., Liu, Y., Dong, J.S., Liu, Y.: Improved BDD-Based Discrete Analysis of Timed Systems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 326–340. Springer, Heidelberg (2012)
15. Ouaknine, J., Worrell, J.: Timed CSP = Closed Timed Safety Automata. Electrical Notes Theoretical Computer Science 68(2) (2002)
16. Palikareva, H., Ouaknine, J., Roscoe, B.: Faster FDR Counterexample Generation Using SAT-Solving. ECEASST 23 (2009)
17. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 133–152. Springer, Heidelberg (1995)
18. Schneider, S.: Concurrent and Real-Time Systems: The CSP Approach. Wiley (2000)
19. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, E.: Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. TOSEM (to appear, 2012)
20. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
21. Sun, J., Liu, Y., Dong, J.S., Zhang, X.: Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 581–600. Springer, Heidelberg (2009)
22. Tripakis, S.: Verifying Progress in Timed Systems. In: Katoen, J.-P. (ed.) ARTS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
23. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS, pp. 332–344. IEEE Computer Society (1986)
24. Wang, F.: Symbolic Verification of Complex Real-Time Systems with Clock-Restriction Diagram. In: FORTE, pp. 235–250 (2001)

Annotations for Alloy: Automated Incremental Analysis Using Domain Specific Solvers

Svetoslav Ganov, Sarfraz Khurshid, and Dewayne E. Perry

Electrical and Computer Engineering,
University of Texas at Austin, Austin TX 78712, USA
svetoslavganov@utexas.edu, {khurshid,perry}@ece.utexas.edu

Abstract. Alloy is a declarative modeling language based on first-order logic with sets and relations. Alloy problems are analyzed fully automatically by the Alloy Analyzer. The analyzer translates a problem for given bounds to a propositional formula for which it searches a satisfying assignment via an off-the-shelf propositional satisfiability (SAT) solver. Hence, the performed analysis is a bounded exhaustive search and increasing the bounds leads to a combinatorial explosion.

We increase the efficiency of the Alloy Analyzer by performing incremental analysis via domain specific solvers. We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. This meta-data is utilized to automatically partition a problem into sub-problems and opportunistically solve independent sub-problems in parallel using dedicated constraint solvers. We integrate dedicated Integer and String constraint solvers into Alloy's SAT based backend. Experimental results show that using dedicated solvers and exploiting independent sub-problems provide better efficiency and scalability; for the chosen subjects, our technique enables up to an order of magnitude speed-up.

1 Introduction

Alloy [1] is a declarative modeling language based on first-order logic with sets and relations. It has been successfully used for identifying problems in semantic models and algorithms [11], detecting anomalous scenarios in security-critical systems [19], automated test generation [14], modeling software architecture [5], etc. Alloy problems are analyzed fully automatically by the Alloy Analyzer. The analyzer translates a problem for given bounds to a propositional formula for which it searches a satisfying assignment via an off-the-shelf propositional satisfiability (SAT) solver. Hence, the performed analysis is a bounded exhaustive search and increasing the bounds leads to a combinatorial explosion.

However, performing analysis within given bounds only guarantees that the obtained results are valid within these bounds. Therefore, increasing the bounds of the analysis would strengthen the confidence in the obtained results. To enable reasoning for increased bounds we focus on improving the speed of Alloy's SAT based backend by exploiting two key ideas: (1) a problem can be decomposed

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 414–429, 2012.

into sub-problems which can be solved incrementally and potentially in parallel; and (2) domain specific solvers enable faster evaluation of problems in their target domain.

The first key idea is that when an Alloy model is translated to SAT, an opportunity to perform a more efficient incremental analysis is not exploited. Incremental reasoning enables reducing the space searched by the solver [17] and enables tackling independent sub-problems in parallel, thus improving performance due to better utilization of contemporary multi-core architectures. For example, generating a binary tree may be performed by generating the structure and using this partial solution to solve in parallel the independent sub-problems of generating the keys, the size, and the parent relationship. To address this we employ an incremental technique for solving Alloy formulas [17], where one solution to a formula provides a partial solution to another formula, which can then be solved more efficiently. We improve this technique by recursively decomposing the problem into the sub-problems, as opposed to decomposing it into only two sub-problems, and opportunistically solve independent sub-problems in parallel.

The second key idea is that when an Alloy formula is translated for the SAT solver, domain specific knowledge is lost, thus an opportunity to take advantage of the problem domain is not exploited. Domain specific solvers are designed for tackling special classes of problems using special representations, and algorithms [6] [12]. For example, finding whether two String variables can be equal is faster by getting the intersection of two automatons than by exploring the cross product of all possible values for the two variables. To address this we introduce *annotations*, an easy-to-use and unobtrusive facility to embed meta-data for mapping Alloy signatures to data types, Alloy predicates to operations on these data types, and bind data types to domain specific solvers. This enables us to opportunistically solve a predicate that depends only on variables of a single data type via the dedicated constraint solver mapped to this data type.

The benefit of our approach is three-fold: (1) incremental analysis limits the search space explored by the solver; (2) opportunistically solving independent sub-problems in parallel improves utilization of system resources; and (3) domain specific solvers are more efficient for problems in their target domain. Experiments show our technique enables better performance and scalability than a SAT-based approach.

This paper makes the following contributions:

- **Incremental analysis with parallel reasoning about independent sub-problems**. We perform incremental analysis of an Alloy problem by recursively decomposing it into sub-problems. We identify likely independent sub-problems and solve them in parallel if they are indeed independent.
- **Domain specific solver integration via annotations**. We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. This enables reasoning about when to use a domain specific solver and how to translate Alloy formulas to the language of that solver.

- **Dedicated solvers for Alloy**. We support a dedicated Integer and a dedicated String constraint solver integrated into Alloy's SAT based backend.
- **Implementation**. We implement our approach into a custom build of the Alloy Analyzer.
- **Evaluation**. We evaluate our approach using small but complex Alloy models, including a model from the standard Alloy distribution. Empirical results show our approach provides up to an order of magnitude speed-up over Alloy's purely SAT-based analysis.

2 Background

In this section we provide background knowledge about Alloy [1] and declarative slicing [17] for incrementally analyzing Alloy models in the context of a binary search tree example.

2.1 Alloy

An Alloy model s can be represented as a triple $< r, s, b >$, where r is the set of relations in s, and b is the bound on the universe of discourse. An *instance*, i.e. a solution, i satisfying an Alloy model is a function from the set of relations r to a power set of tuples 2^T where each tuple consists of indivisible atoms, i.e., $i : r \rightarrow 2^T$. Hence, an instance gives the set of tuples that valuate every relation. The *canonical form* of an Alloy model is $\wedge p_i$, for $i = 1, ..., n$, where each p_i is an arbitrary formula.

2.2 Alloy Model - Binary Search Tree Example

A binary search tree is a node-based data structure where: (1) each node has at most two children–left and right–whose parent is the given node; (2) the left sub-tree rooted at a given node contains keys less than the key of that node; (3) the right sub-tree rooted at a given node contains keys greater than the key of that node; and (4) the left and right sub-trees are also binary search trees; In Fig. 1 is depicted the Alloy model for a binary search tree.

First, we declare the entities contained in the model. A node (line 3) has: (1) at most one left child (line 4); (2) at most one right child (line 5); (3) at most one parent (line 6); and (4) a key (line 7); A binary tree (line 9) has: (1) at most one node as its root (line 10); and (2) a size (line 11);

Next, we specify the relationships between the model entities to reflect the properties of the data structure. A binary search tree is *Acyclic* (line 13), which is for every node reachable from the root performing zero or more traversals (line 14): (1) at most one node is visited following the left and right relations in reverse direction (line 15); (2) a node cannot be reached by following one or more times the left and right relations beginning from that node (line 16); and (3) the left and right nodes are disjoint (line 17);

```
1   module BinarySearchTree
2
3   sig Node {
4     left:lone Node,
5     right:lone Node,
6     parent:lone Node,
7     key:Int
8   }
9   sig BinarySearchTree {
10    root:lone Node,
11    size:Int
12  }
13  pred Acyclic(t:BinarySearchTree) {
14    all n:t.root.*(left+right) {
15      lone n.~(left+right)
16      n !in n.^(left+right)
17      no n.left & n.right
18    }
19  }
20  pred Parent(t:BinarySearchTree) {
21    all n,n':t.root.*(left+right) | n in n'.(left+right) => n' = n.parent
22    no t.root.parent
23  }
24  pred Search(t:BinarySearchTree) {
25    all n:t.root.*(left+right) {
26      all n':n.left.*(left+right) | int n'.key < int n.key
27      all n':n.right.*(left+right) | int n.key < int n'.key
28    }
29  }
30  pred Size(t:BinarySearchTree) {
31    int t.size = #(t.root.*(left+right))
32  }
33  pred BinarySearchTree(t:BinarySearchTree) {
34    Acyclic[t] && Parent[t] && Search[t] && Size[t]
35  }
36  run BinarySearchTree exactly 1 BinarySearchTree, exactly 3 Node
```

Fig. 1. Alloy model of a binary search tree

The nodes in the binary search tree have a *Parent* property (line 20), which is: (1) every node reachable from the root performing zero or more traversals is the parent of its left and right children (line 21); and (2) the root has no parent (line 22);

A binary search tree contains data satisfying the *Search* (line 24) property, which is for every node reachable from the root performing zero or more traversals (line 25): (1) every descendant reached by following the left and right relations of its left child zero or more times has a lesser key (line 26); and (2) every descendant reached by following the left and right relations of its right child zero or more times has a greater key (line 27);

A binary search tree has a *Size* property (line 30), which is the cardinality of the nodes reached from its root by performing zero or more traversals of the left and right relations (line 31).

In order for a data structure to be a *BinarySearchTree* (line 33) it has to satisfy the *Acyclic*, *Parent*, *Search*, and *Size* predicates (line 34).

Finally, we request from the analyzer to find an instance by specifying bounds on the cardinality of atoms (line 36). Upon running this command the analyzer tries to find valuations to the relations such that the predicate declaring a binary search tree evaluates to true, which is an instance that satisfies the model exists.

```
1   // free variables: {root, left, right}
2   all n:t.root.*(left+right) | lone n.~(left+right)
3   // free variables: {root, left, right}
4   all n:t.root.*(left+right) | n !in n.^(left+right)
5   // free variables: {root, left, right}
6   all n:t.root.*(left+right) | no n.left & n.right
7   // free variables: {root, left, right, parent}
8   all n,n':t.root.*(left+right) | n in n'.(left+right) => n' = n.parent
9   // free variables: {root, parent}
10  no t.root.parent
11  // free variables: {root, left, right, key}
12  all n:t.root.*(left+right) {
13    all n':n.left.*(left+right) | int n'.key < int n.key }
14  // free variables: {root, left, right, key}
15  all n:t.root.*(left+right) {
16    all n':n.right.*(left+right) | int n.key < int n'.key }
17  // free variables: {root, left, right, size}
18  int t.size = #(t.root.*(left+right))
```

Fig. 2. Normalized form of the constraints in the binary search tree model

2.3 Declarative Slicing - Binary Search Tree Example

Declarative slicing [17] in the context of an Alloy model in a canonical form - $\wedge p_i$, for $i = 1, ..., n$, where each p_i is an arbitrary formula - is to partition an Alloy model s into a *base* slice s_b and a *derived* slice s_d. The base and the derived slices consist of disjoint subsets of the model constraints. The base slice is derived via a *slicing criterion* c which is a subset of the model relations r, i.e. $c \subseteq r$. The base slice contains the constraints that involve only relations in the slicing criterion, i.e., $s_b : \wedge q_i$ for $i = 1, \ldots, m$, where each $q_i \in \{p_1, \ldots, p_n\}$ and $free\_variables \cap predicate\_variables(q_i) \subseteq c$. The rest of the model constraints belong to the derived slice, i.e. $s_d : \wedge d_i$ for $i = 1, \ldots, t$, where each $d_i \in \{p_1, \ldots, p_n\}$ and $free\_variables \cap predicate\_variables(d_i) \not\subseteq c$. Once the model is partitioned into a base and a derived slice, a solution for the base slice is extended into a solution for the entire model.

The first step in the declarative slicing technique is model normalization during which composite constraints (e.g. nested quantified formulas) are partitioned, i.e. the model is translated to a canonical form. The normalized form of the binary search tree model from Section 2.2 is presented in Fig. 2.

The second step is choosing an optimal slicing criterion, since more than one such may exist, by using each possible slicing criterion to analyze the model for a smaller bounds and based on some metrics selecting the one that is likely to provide the most significant speed up. The possible slicing criteria are ordered under set containment where each slicing criterion represents a free variable combination from some model constraints. The possible slicing criteria for the binary search tree are presented in Fig. 3.

The third step is solving the problem for the desired bounds using the optimal slicing criterion from the previous step. It is possible that a solution for the base slice cannot be extended to a solution for the entire problem since the derived slice may additionally constrain the relations in the slicing criterion. Therefore, all possible solutions for the base slice are attempted to be extended to a complete

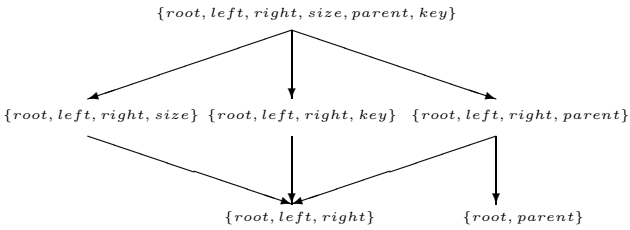


Fig. 3. Partially ordered set of slicing criteria

solution. If a solution for the entire problem is found, the problem is reported consistent, otherwise the problem is declared inconsistent for the given bounds.

3 Our Approach

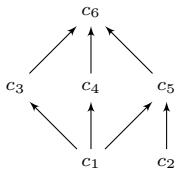
In this section we present our approach for incremental analysis of Alloy problems via domain specific solvers. We exploit two key observations: (1) a problem can be decomposed into sub-problems which can be solved incrementally and potentially in parallel; and (2) domain specific solvers enable faster evaluation of problems in their target domain.

3.1 Incremental Analysis with Parallel Reasoning

When an Alloy model is translated to SAT, an opportunity for a more efficient incremental analysis is not exploited. Previous work [17] introduced an approach to decompose the problem into a base and a derived slice, and extend a solution for the base to one for the entire problem. This approach improves traditional analysis but there are two areas for enhancement: (1) the problem is partitioned only into two sub-problems; and (2) the two sub-problems are solved sequentially.

Instead of partitioning the problem into two sub-problems, as the declarative slicing technique presented in Section 2.3, we partition the problem into multiple sub-problems. We apply the partitioning procedure for declarative slicing recursively. We select the smallest slicing criterion, i.e. the criterion with the minimum number of free variables, and partition the problem into a base and a derived slice. Then we select the next smallest slicing criterion and partition the derived slice from the previous step into a base and a derived slice. We repeat the procedure until the slicing criterion includes all free variables.

Recall that the possible slicing criteria for declarative slicing can be envisioned as a partially ordered set under set containment which is represented as a join-semi-lattice. We use that partially ordered set of slicing criteria to construct a dependency graph $G = (V, E)$ with a set of vertices V and a set of edges E as follows: (1) add a vertex $v_i \in V$ for every slicing criterion c_i where $i \in (1, \ldots, n)$; and (2) add a directed edge $e_{ij} \in E$ from the vertex representing criterion c_i to the vertex representing criterion c_j if the slicing criterion c_j is the smallest slicing criterion that contains c_i, i.e. $c_i < c_j \wedge \nexists c_k : c_i < c_k < c_j \rightarrow e_{ij} = (v_i, v_j)$



$c_1 = \{root, left, right\}$
$c_2 = \{root, parent\}$
$c_3 = \{root, left, right, size\}$

$c_4 = \{root, left, right, key\}$
$c_5 = \{root, left, right, parent\}$
$c_6 = \{root, left, right, size, parent, key\}$

Fig. 4. Slicing criteria dependency graph

where $i \neq j \neq k$ and $i, j, k \in (1, \ldots, n)$. The dependency graph for the binary search tree from Section 2.2 is shown in Fig. 4.

Once we have constructed the dependency graph, we perform a topological sort of the nodes in that graph to obtain an ordering of the slicing criteria used for incremental analysis. For example, a topological ordering of the nodes in the dependency graph from Fig. 4 is the sequence $< c_1, c_2, c_3, c_4, c_5, c_6 >$. This slicing criteria sequence is used to incrementally analyze the model.

We slice the model based on the first slicing criterion from the sequence, i.e. select constraints that involve only relations from the slicing criterion. Then we solve the constraints in the slicing criterion. If a solution is found, we slice the model based on the second slicing criterion. Before solving the constraints for current base we set the relations in these constraints to valuations we found in the previous step (e.g. we set a partial solution). We now solve the constraints. This procedure is repeated until a solution for the entire problem is found or we determine that the problem is inconsistent.

Since the constraints in the current iteration may constrain relations whose valuations were found in a previous one (e.g. such relations were under constrained in the previous iteration), it is possible that a solution for these constraints cannot be found. In such a case we backtrack to the sub-problem we solved in the previous iteration for which we try to find another solution which is then propagated to the current sub-problem and a new attempt to solve the current sub-problem is made. In case no solution for the previous sub-problem can be extended to a solution for the current one, we declare the previous sub-problem inconsistent and try to backtrack to the sub-problem preceding it. If no solution for the first sub-problem can be extended to a solution for the entire problem, we declare the problem inconsistent.

Note that a subsequence of slicing criteria in the topologically sorted sequence may share no common relations. In such a case we solve the sub-problems for each slicing criterion in parallel. It is also possible that a subsequence of slicing criteria share common relations but none of the predicates that belong to their corresponding slices imposes constraints on the common relations. Hence, sub-problems resulting from slicing the model based on a subsequence of independent, i.e. with no edge in the dependency graph, slicing criteria *may* be independent.

```
1  Solution solve(Problem problem) {
2    // Try solving potentially parallel problems.
3    List bases = getParallelBases(problem);
4    Solution solution = solveParallel(bases);
5    if (solution.isValid()) {
6      Problem slice = problem - bases;
7      if (slice == null) {
8        return solution;
9      }
10     slice.setPartialSolution(solution);
11     solution = solve(slice);
12     if (solution.isValid()) {
13       return solution;
14     }
15   }
16   // Try extending a base solution to a full one.
17   Problem base = getBase(problem);
18   Problem slice =  problem - base;
19   for (Solution solution in solveAll(base)) {
20     if (slice == null) {
21       return solution;
22     }
23     slice.setPartialSolution(solution);
24     solution = solve(slice);
25     if (solution.isValid()) {
26       return solution;
27     }
28   }
29   return Solution.INVALID;
30 }
```

Fig. 5. Incremental analysis with parallel reasoning algorithm

```
1   module Integer
2
3   . . .
4
5   @datatype(solver="IntegerSolver")
6   sig integer {
7       value: int
8   }
9
10  @operation(name="lessThan")
11  pred integerLessThan(lhs, rhs: int) {
12      lhs.value < rhs.value
13  }
14
15  . . .
16
```

Fig. 6. Sample of the Annotated Integer module

```
1  for (BinaryTree t:eval(BinaryTree)) {
2    for (Node n:eval(t.root.*(left+right))) {
3      for (Node n':eval(n.root.*(left+right))) {
4        IntVar lhs = Solver.newVar(
5            t.name() + n'.name());
6        IntVar rhs = Solver.newVar(
7            t.name() + n.name());
8        Solver.lessThan(lhs, rhs);
9      }
10   }
11 }
```

Fig. 7. Constraint translation for a dedicated Integer solver

In such a case we try to solve such sub-problems in parallel for small bounds and, in case we succeed, we solve the sub-problems in parallel for the current bounds. The described algorithm is presented in Fig. 5.

In particular, for the binary search tree model our algorithm performs an incremental analysis based on the following slicing criteria $c_1 \rightarrow c_2 \rightarrow parallel(c_3, c_4, c_5) \rightarrow c_6$, where the predicates corresponding to slicing criteria c_3, c_4, and c_5 are solved in parallel. Note that our algorithm on Fig. 5 is doing a best effort for solving as many sub-problems as possible in parallel. In case the solutions for the parallel sub-problems are conflicting (due to additional constraints on the common relations), we perform systematic exploration of each slicing criterion one at a time.

3.2 Dedicated Solver Integration via Annotations

When an Alloy problem is translated to propositional logic, domain specific knowledge is lost. However, knowledge about the problem domain creates an opportunity of using domain specific solvers–solvers designed for tackling special classes of problems via special representations and algorithms. Performing an incremental analysis of an Alloy model creates an opportunity for employing domain specific solvers for relevant sub-problems, i.e. slices. This opportunity has

been recognized by previous work on declarative slicing [17]. However, there are three limitation of this work that we address: (1) there is no generic mechanism for integrating domain specific solvers into the Alloy engine; (2) a mechanism for determining when a domain specific solver can be used is lacking; and (3) analysis only with an Integer domain specific solver has been presented.

We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. These annotations allow us to determine when to use a given domain specific solver as well as facilitates the translation from Alloy to the language of the specialized solver.

The *@datatype* annotation can be placed only on a signature definition and specifies a mapping from an Alloy signature to a domain specific data type. This annotation has one attribute, *solver*, that specifies which solver to use for reasoning about constraints over the annotated signature (the data type is inferred by the solver). We also define an annotation *@operation* that can be placed only on a predicate and specifies to which domain specific operation to map the predicate (the solver is inferred from the type of the arguments). An annotated model for an Integer type which serves as a wrapper around the built-in *int* to enable annotation is presented in Fig. 6.

We use the meta-data from the annotations and *evaluation* of an expression against a given instance (solution) supported by Alloy's backend Kodkod [16] to determine whether a domain specific solver may be used for solving the constraints in the currently analyzed slice and how to translate these constraints to the language of the dedicated solver. We illustrate this with an example.

Assume we already have a solution of a binary search tree with three nodes for the previous slicing criterion $\{root, left, right\}$ and the current slicing criterion is $\{root, left, right, key\}$ for which we have set the partial solution from the previous slice. We traverse the constraints in the current slice, and for each of them: first check via an evaluation whether it is already satisfied and, if not, what free variables it constrains. If all unsatisfied constraints constrain free variables of the same type, we can use the dedicated solver specified in the *@datatype* annotation, otherwise we fall back to SAT. In the example, all constraints for slicing criterion $\{root, left, right, key\}$ constrain only the free variable *key* which is of type Integer annotated to use the *IntegerSolver*. Hence, we can use the dedicated Integer constraint solver.

Now we have to translate the constraints in the slice to the language of the solver. Without loss of generality consider the constraint $one\ t : BinaryTree\ \&\&\ all\ n : t.root. * (left + right)\{all\ n' : n.left. * (left + right) \mid integerLessThan\ (n'.key, n.key)\}$ (added declaration of t for clarity and changed key to our custom Integer type). We evaluate the variable t from declaration $one\ t : BinaryTree$ which returns $\{BinaryTree\$0\}$. Hence, for some binary tree in this set the constraint must hold. We next evaluate the variable n from the next declaration $n : t.root. * (left + right)$ for every t which returns $\{Node\$0, Node\$1, Node\$2\}$. Follows evaluation of the variable n' from the next declaration $n' : n.left. * (left + right)$ for every n which returns $\{Node\$1, Node\$2\}, \{Node\$2\}, \{\}$. Now we have identified all nodes whose keys are constrained and we can use them

to construct a problem in the domain of the Integer solver. We add a variable for the key of each node and a constraint for every pair on nodes constrained by $integerLessThan(n'.key, n.key)$. Note that the predicate $integerLessThan$ is mapped to the $lessThan$ domain specific operation. A translation of the constraint to the language of the Integer constraint solver is presented in Fig. 7. For solving integer constraints we use Choco [4].

In addition to the dedicated Integer constraint solver we provide a specialized solver for String constraints. Similarly, we define a custom module for the String data type to which we apply the corresponding annotations. Since existing off-the-shelf String solvers [6,12] do not support multi-variable problems, we have implemented a String constraint solver that supports multi-variable problems with binary constraints. We use recursive backtracking search with constraint propagation, via the AC-3 [13] algorithm, applying the *minimal remaining values* and *degree* heuristics to guide the search. We use finite state automata [3] to represent variable domains and performing operations on them.

Adding other solvers can be done similarly to the Integer and String ones. An Alloy module, i.e. a model file, with operation mappings has to be written and a JAR file with the solver implementation has to be deployed. No changes to the Alloy source are required. Hence, we provide a general mechanism for adding domain specific solvers to the Alloy's SAT based backend.

4 Evaluation

We have evaluated our approach on several data structure models with Integer and String data and a P2P protocol model. Each model was analyzed with the conventional Alloy Analyzer (4.1.10) and a version that incorporates our technique. We report analysis results in terms of solving time for given bounds and maximal bounds reached within reasonable time.

The evaluation system was a laptop with an Intel i7 M620 2.67GHz processor, 4GB of RAM running Ubuntu 10.04 Lucid. For all experiments the analyzer was set to use 4096MB of memory and 65536K maximal stack size. All tests were run on a cold VM to avoid just-in-time compilation or cache skewing the results.

An analysis command takes as arguments a bund for each signature, recall the command for the binary search tree model in Fig 1 (line 46). We have set these parameters as follows: (1) the bound for the signature representing the modeled data structure or state for the protocol was always one; and (2) the bound for the signature representing nodes of the data structure or the protocol was incrementally increased.

We have also evaluated our technique on a model of the Chord P2P protocol [2] which is one of the sample models posted on the Alloy home page. Note that the model does not incorporate Integer or String constraints which precludes the use of multiple solvers. However, the model can be analyzed incrementally. The results of our analysis are presented in Table 5 and Fig. 12. The data in the table and the figure are arranged similarly to the ones for already presented data structures. Note that our technique is twice as fast for ten nodes as opposed

Table 1. Sorted linked list with Integer data

Node count	2	4	6	8	10	12	14	16	20	30
Standard Analyzer (ms)	111	335	896	2401	15695	58307	124335	N/A	N/A	N/A
Incr. SAT solver (ms)	202	372	822	1251	1743	2460	4197	4131	52256	252572
Incr. multi-solver (ms)	327	412	781	1244	1251	1760	3179	2971	46817	234658

Table 2. Sorted linked list with String data

Node count	2	4	6	8	10	12	14	16	20	24
Standard Analyzer (ms)	1763	3276	7293	69851	147406	294445	N/A	N/A	N/A	N/A
Incr. SAT solver (ms)	2158	2702	6168	4688	4738	19822	81546	58312	160286	N/A
Incr. multi-solver (ms)	673	984	1358	1515	2280	2516	3841	6602	92722	183440

Table 3. Binary search tree with Integer data

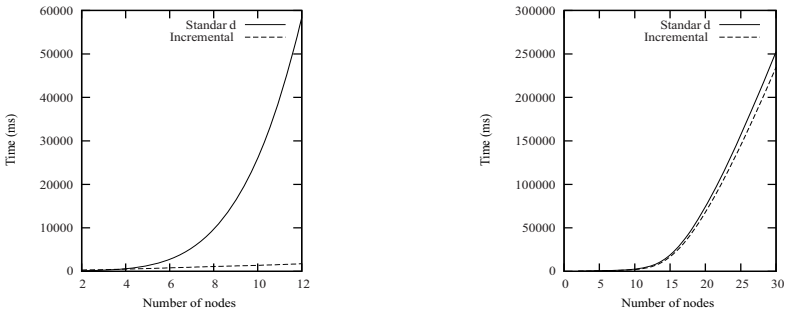
Node count	2	4	6	8	10	20	30	40
Standard Analyzer (ms)	116	322	1174	8724	N/A	N/A	N/A	N/A
Incr. SAT solver (ms)	260	414	1076	1522	2546	9400	59527	N/A
Incr. multi-solver (ms)	368	422	990	1317	1534	4935	29297	166369

Table 4. Binary search tree with String data

Node count	2	4	6	8	10	20	30	40
Standard Analyzer (ms)	2307	5571	14206	127185	1241175	N/A	N/A	N/A
Incr. SAT solver (ms)	2464	3422	5181	6422	24150	94048	N/A	N/A
Incr. multi-solver (ms)	1069	1424	1752	2090	2815	13389	40847	1020314

Table 5. Chord P2P protocol

Node count	2	4	6	8	10
Standard Analyzer (ms)	15	89	792	7739	127987
Incr.SAT solver (ms)	52	122	708	5736	60139

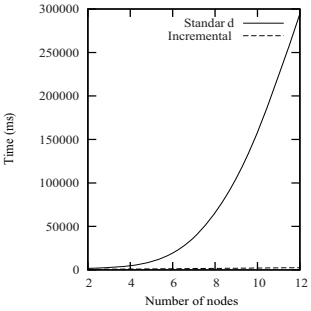


(a) Standard vs incr. multi-solver (b) Incr. SAT vs incr. multi-solver

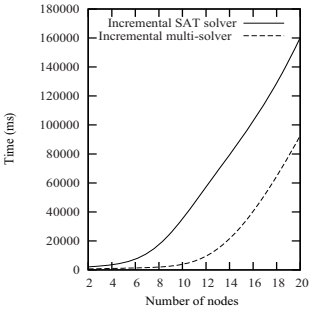
Fig. 8. Sorted linked list with Integer data

to the standard analysis. This example demonstrates that, even if employing multiple solvers is not feasible due to the model nature, performing incremental reasoning leads to an improvement in terms of analysis speed.

Our results indicate that for small bounds our incremental multi-solver analysis is as fast as the standard Alloy Analyzer but, as the bounds grow, the gains

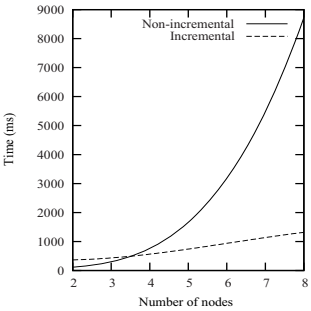


(a) Standard vs incr. multi-solver

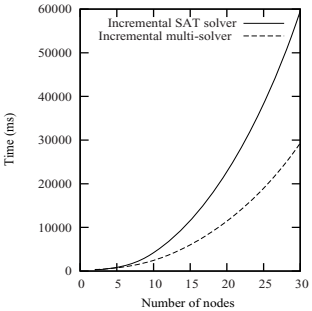


(b) Incr. SAT vs incr. multi-solver

Fig. 9. Sorted linked list with String data

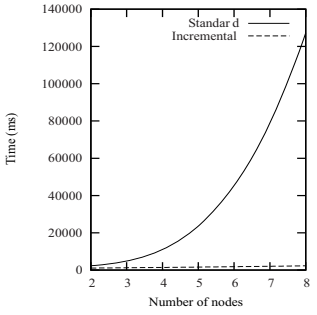


(a) Standard vs incr. multi-solver

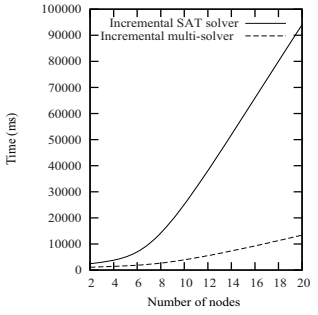


(b) Incr. SAT vs incr. multi-solver

Fig. 10. Binary search tree with Integer data



(a) Standard vs incr. multi-solver



(b) Incr. SAT vs incr. multi-solver

Fig. 11. Binary search tree with String data

of using our technique become significant. This can be explained with the linearly growing costs of problem translation and multi-solver initialization which is amortized over an exponentially growing search space. Our approach is almost two orders of magnitude faster (except for the binary search tree with Integer

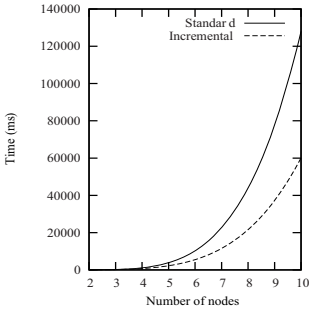


Fig. 12. Chord P2P protocol model - standard vs incremental SAT

data and the Chord P2P model) for the bounds reachable by the standard analyzer. Further, even if a model is not suitable for employing multiple solvers, it can be solved incrementally, increasing analysis speed.

In Table 1 are presented the results for a sorted linked list with Integer data. The first row is the number of nodes, the second row is solving time via the standard Alloy Analyzer, the third row is solving time via incremental analysis using a SAT solver, and the fourth row is solving time via incremental analysis using multiple solvers. Note that for certain scopes no results are reported because either the analysis took more than thirty minutes or the solver ran out of memory. In Fig. 8(a) is presented the improvement in terms of solving time for incremental multi-solver analysis as opposed to the standard Alloy Analyzer and in Fig. 8(b) is depicted the improvement in terms of solving time for incremental multi-solver as opposed to incremental SAT analysis. We are using Bezier curves in all figures to depict smoothened trends rather than local fluctuations.

We present results for a sorted linked list with String data (Table 2, Fig. 9(a), Fig. 9(b)), a binary search tree with Integer data (Table 3, Fig. 10(a), Fig. 10(b)), and a binary search tree with String data (Table 4, Fig. 11(a), and Fig. 11(b)) similarly to the ones for a sorted linked list with Integer data.

5 Related Work

This paper introduces annotations for Alloy models that define data types, operations on these data types, and bindings from data types to domain specific solvers. We utilize this information to automatically partition the problem into multiple sub-problems. We also employ this meta-data to determine when to use a dedicated solver as well as to facilitate translation of the solved sub-problem into the language of the domain specific solver. Additionally, we integrate an Integer constraint solver and a String constraint solver into Alloy's backend.

In a recent publication [8] we have proposed the idea of using annotations in Alloy for guiding problem partitioning. In particular, explicitly specifying the priority of a predicate as well as the solver to be used for its analysis. In the current work we present an approach to automatically partition the problem into

multiple sub-problems some of which are solved opportunistically with domain specific solvers. In this paper we introduce annotations that define data types, mapping operations on such types to their domain specific counterparts, and mappings from data types to dedicated constraint solvers.

Incremental solving for Alloy models, where a solution to one formula is fed as a partial solution to solve another formula, was introduced by Uzuncaova et al. [17,18] in the context of test input generation for software product lines. This work partitioned the problem into two sub-problems while our technique aims to maximize the number of partitions. Further, this work does not specify an algorithm for determining when it is possible to use a dedicated solver and we provide such a technique. This work solves the two sub-problems in sequence but we try to opportunistically parallelize analysis of independent sub-problems. We also have introduced a dedicated String constraint solver for Alloy.

The second author co-authored a recent paper [10] that introduces *mixed constraints*, which are written using a combination of a declarative language (Alloy), and an imperative language (Java). It supports annotating *def-use* sets of variables to facilitate solving of mixed constraints using different solvers, where each solver is designed for constraints written using one particular paradigm. Mixed constraints offer a complementary approach to this paper. They facilitate writing of constraints using a combination of declarative and imperative paradigms, whereas this paper focuses on efficient solving of models written purely in Alloy, hence does not require learning a new notation.

Parallel analysis of Alloy models was first proposed in [15]. This work explores providing a parallel SAT solver with Alloy specific enhancements by partitioning the propositional formula among parallel SAT solvers. In contrast, we partition the problem into sub-problems before it has been translated to propositional logic enabling the use of domain specific solvers. We also identify sub-problems that may be solved in parallel, some via a SAT solver.

An example of extending Alloy's syntax to describe dynamic properties of systems via actions is presented in [7]. The actions enable specifying dynamic properties of execution traces as dynamic logic specifications. Our technique is similar with respect to extending the Alloy syntax with new semantic features and implementing automated analysis of the latter. While this work focuses on adding constructs for specifying dynamic behavior, we embed meta-data in a standard Alloy model to achieve scalable and efficient analysis.

An approach of using SAT Modulo Theories solver (SMT) for analyzing Alloy specifications is presented in [9]. The SMT solver does not replace the Alloy SAT-based back-end, rather complements it by potentially detecting if a formula is a tautology, a capability the Alloy Analyzer is lacking. This does not require finitizing the values for each relation. Similarly, we introduce a set of dedicated solvers for augmenting Alloy's analysis backend. However, we are using domain specific solvers and perform an incremental analysis. Our technique is to partition the problem and solve it in finitized bounds with specialized solvers.

6 Conclusion

We increase the efficiency of the Alloy Analyzer by performing an incremental analysis via domain specific solvers. We introduce annotations that define data types, operations on these data types, and bindings from data types to domain specific solvers. This meta-data is utilized to opportunistically solve a sub-problem using a dedicated constraint solver. Our technique automatically partitions the problem into sub-problems and opportunistically solves independent sub-problems in parallel. We integrate a dedicated Integer constraint solver and a String constraint solver in Alloy's SAT based backend.

We have evaluated our approach on selected data structure models with both Integer and String data and one P2P protocol. Our technique achieves a substantial increase in analysis speed, thus enabling us to reach greater bounds. We believe that annotations have an important role to play in analysis of declarative programs as well as in the context of other analyzers such as model checkers and theorem provers.

Acknowledgement. This work was funded in part by the National Science Foundation under Grant Nos. IIS-0438967, CCF-0828251, CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

1. Alloy Analyzer 4 (March 2011), http://alloy.mit.edu/alloy4/
2. Alloy model of Chord (May 2011), http://alloy.mit.edu/community/files/chord.pdf
3. Automaton library (March 2011), http://www.brics.dk/automaton/
4. Choco constraint solver (March 2011), http://www.emn.fr/z-info/choco-solver/
5. Auguston, M.: Software architecture built from behavior models. SIGSOFT Softw. Eng. Notes 34(5), 1–15 (2009)
6. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise Analysis of String Expressions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 1–18. Springer, Heidelberg (2003)
7. Frias, M.F., Galeotti, J.P., López Pombo, C.G., Aguirre, N.M.: DynAlloy: upgrading Alloy with actions. In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 442–451. ACM, New York (2005)
8. Ganov, S., Khurshid, S., Perry, D.: A case for Alloy annotations for efficient incremental analysis via domain specific solvers. In: 26th IEEE/ACM International Conference on Software Engineering, Lawrence, Kan, USA (2011)
9. El Ghazi, A.A., Taghdiri, M.: Relational Reasoning via SMT Solving. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 133–148. Springer, Heidelberg (2011)
10. Khalek, S., Narayanan, V., Khurshid, S.: Mixed constraints for test input generation - an initial exploration. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 548–551 (November 2011)
11. Khurshid, S., Jackson, D.: Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE 2000 (2000)

12. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: a solver for string constraints. In: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, pp. 105–116. ACM, New York (2009)
13. Mackworth, A.K., Freuder, E.C.: The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. Artif. Intell. 25(1), 65–74 (1985)
14. Marinov, D., Khurshid, S.: TestEra: A novel framework for automated testing of Java programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, ASE 2001 (2001)
15. Rosner, N., Galeotti, J.P., Lopez Pombo, C.G., Frias, M.F.: ParAlloy: Towards a Framework for Efficient Parallel Analysis of Alloy Models. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 396–397. Springer, Heidelberg (2010)
16. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
17. Uzuncaova, E., Khurshid, S.: Constraint Prioritization for Efficient Analysis of Declarative Models. In: Cuellar, J., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 310–325. Springer, Heidelberg (2008)
18. Uzuncaova, E., Khurshid, S., Batory, D.S.: Incremental test generation for software product lines. IEEE Trans. Software Eng. 36(3), 309–322 (2010)
19. Woodcock, J., Aydal, E.G., Chapman, R.: The Tokeneer experiments. In: Roscoe, A., Jones, C.B., Wood, K.R. (eds.) Reflections on the Work of C.A.R. Hoare. History of Computing, pp. 405–430. Springer, London (2010)

State Space c-Reductions of Concurrent Systems in Rewriting Logic*

Alberto Lluch Lafuente[1], José Meseguer[2], and Andrea Vandin[1]

[1] IMT Institute for Advanced Studies Lucca, Italy
[2] University of Illinois in Urbana-Champaign, USA

Abstract. We present *c-reductions*, a simple, flexible and very general state space reduction technique that exploits an equivalence relation on states that is a bisimulation. Reduction is achieved by a *canonizer function*, which maps each state into a not necessarily unique canonical representative of its equivalence class. The approach contains symmetry reduction and name reuse and name abstraction as special cases, and exploits the expressiveness of rewriting logic and its realization in Maude to automate c-reductions and to seamlessly integrate model checking and the discharging of correctness proof obligations. The performance of the approach has been validated over a set of representative case studies.

1 Introduction

Taming state space explosion is one of the key challenges for effective model checking analysis. Bisimulation-based state space reductions are particularly attractive, because they *never generate spurious behaviors*. This is because temporal logic properties are preserved by bisimulations. Therefore, an LTL, CTL, or CTL* formula holds on a bisimilar reduced system iff it holds in the original system. A particular example are *symmetry reductions* which have been extensively studied [1] and are used in model checkers (from the seminal works on Murphi to extensions of Spin, Uppaal, Prism, etc.) and other verification tools such as SAT solvers or planners. Developing and applying such state space reduction techniques is still a challenging task: (i) automatic detection of system regularities like symmetries is not trivial and thus often delegated to the system designer; (ii) their exploitation is sometimes done by enriching the system description language (e.g. *scalarset* datatypes in [2, 3]), so that the user is required to learn new primitives; (iii) the implementation of state space reduction techniques has to be combined (both theoretically and practically) with the rest of the techniques and algorithms implemented in the model checker, and often this integration effort has to be repeated for every new version, improvement or technique; and (iv) checking correctness of the reductions is not easy and requires reasoning techniques (e.g. theorem proving) that may not be integrated in the model checking framework, or part of the user's skills. Indeed, problem

* Work supported by NSF Grant CCF 09-05584, AFOSR Grant FA8750-11-2-0084 and the EU Project ASCENS.

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 430–446, 2012.

(iv) means that in order to correctly model check a formula in a reduced system it must be a correct reduction of the original system, which requires discharging *proof obligations*. The problem, however, is that most model checkers lack theorem proving support (within the same framework) for discharging such proof obligations, so that the checking task is usually left to the user and may never be done, decreasing the confidence that can be placed on the verification.

Research Questions. In addressing problems (i)–(iv) above, our work asks and provides answers to the following research questions: (1) Can symmetry reductions be generalized to reductions *requiring only that the bisimulation is an equivalence relation*? (2) Can model checking support for such bisimulation-based reductions be provided in a way that does *not* require any changes to the underlying model checker, yet with high performance? (3) Can the system description language be kept likewise unchanged? (4) Can the specifications of *reduced systems* be automatically generated from those of the original systems? (5) Can model checking and theorem proving be *seamlessly integrated* for such reductions, so that correctness proof obligations are explicitly generated and can be semi-automatically discharged by appropriate tools?

Our Contributions. We answer question (1) in the affirmative by proposing the notion of c-*reduction*, based on the idea of providing a *canonizer function* that computes a not-necessarily unique representative of the equivalence class of states defined by the bisimulation. This notion is quite flexible, since unique canonical representatives, although maximally space-efficient, can be time-inefficient. Furthermore, it is *fully general*: it subsumes various reduction techniques such as symmetry reduction, name reuse and name abstraction; and it can be applied to any Kripke structure. Questions (2) and (3) are answered in the affirmative: no such changes are needed (moreover, in [8] we report on performance experiments showing that c-reductions can achieve drastic state space reductions). Question (5) is answered by proposing rewriting logic [4] as an efficiently executable *logical framework* supported by a high-performance tool (Maude [5]) and having a formal tool environment where both LTL model checking and the discharging of correctness proof obligations for c-reductions are seamlessly integrated and partially automated. In fact, our answer to question (5) takes the form of a *formal methodology*, which breaks proofs of correctness into smaller, manageable proof subtasks. Many of the steps in our methodology apply to any c-reduction, but some of them are directly tailored to symmetry reductions. As we gain more experience, we plan to extend all steps of our methodology to arbitrary c-reductions. Question (4) is answered in our current prototype for a very wide class of concurrent systems, namely, *object-based concurrent systems*, and takes the form of a *theory transformation* that automatically maps the original system into the desired c-reduction of it.

We have evaluated our approach over a set of examples by considering the ease of defining reduction strategies, the effectiveness of the correctness checks, and the performance of the resulting reductions. Compared to previous work, we have observed performance gains in some cases (including previous implementations of symmetry reductions in Maude [6]), and a great flexibility in the

definition of reductions, which allows us to subsume a wide range of reductions including permutation and rotation symmetries, name reuse and name abstraction, which have interesting applications (e.g. implementation of the operational semantics of languages with dynamic features such as resource allocation). The usefulness of our proof methodology has also been evaluated through a case study. A preliminary version of our tool is available for download [7].

Synopsis. Sect. 2 offers the necessary background. Sect. 3 presents c-reductions in a generic way, focusing on Kripke structures. Sect. 4 describes the realization of c-reductions in rewriting logic, highlighting the theoretical results, and the reasoning and verification mechanisms and tools underlying our methodology for specifying and verifying c-reductions. Sect. 5 covers related work and conclusions.[1]

2 Preliminaries

We will use a simple running example of a banking system[2] of concurrent objects of the same class (accounts) having a natural number as attribute (their balance), and body-less messages (one dollar transfers) for them. The behavior of objects is governed by a simple rule: a message m for an object i can be consumed by object i to increment its balance by one. The system exhibits a clear symmetry: all objects are instances of the same class and have the same behaviour.

Systems like this (and of course more sophisticated ones) can be easily specificied as theories of rewriting logic [4], which can be specified as Maude [5] modules to be executed and analyzed within the Maude framework.

Definition 1 (rewrite theory). *A* rewrite theory $\mathcal{M}$ *is a tuple* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *where* Σ *is a signature, specifying the basic syntax (function symbols) and type infrastructure (sorts, kinds and subsorting) for terms, i.e. state descriptions;* E *is a set of (possibly conditional) equations, which induce equivalence classes of terms (and are used to specify functions), and (possibly conditional) membership predicates, which refine the typing information;* A *is a set of axioms which also induce equivalence classes of terms, i.e. equational axioms describing structural equivalences between terms, like associativity and commutativity;* R *is a set of (possibly conditional) non-equational rules, which specify the local concurrent transitions in a system whose states are* $E \cup A$*-equivalence classes of ground* Σ*-terms; and where* $\phi : \Sigma \to \mathcal{P}_{fin}(\mathbb{N})$ *is a* frozenness map*, assigning to each function symbol* f *of arity* n *a subset* $\phi(f) \subseteq \{1..n\}$ *of its frozen argument positions, i.e. positions under which rewriting with rules in* R *is forbidden.*

In our example we can define a theory (a Maude module) `BANK` whose signature Σ includes sorts for messages (`Message`), objects (`Object`), their identifiers

[1] Interested readers are referred to [8] which includes complementary material: the formal proofs ([8, Sect. A]), a performance evaluation with literature benchmark ([8, Sect. B]), and a full description of our case study ([8, Sect. C,D]).

[2] Indeed, it is a simplification of the model of a bank account system described in [5].

and attributes as natural numbers (`Nat`), configurations (`Configuration`) and states (`State`), and operators that allow us to represent an object `i` with attribute `x` as a term `< i | x >`, a message for object `i` as a term `credit(i)`, an empty configuration (of objects and messages) by `none`, and the multiset union of configurations by juxtaposition, obeying associativity and commutativity as axioms. The operator {_} wraps an entire configuration `c` as a state {`c`}. Rules `rl { < i | x > credit(i) c1 } => { < i | s(x) > c1}` , and `rl { < i | x > credit(i) } => { < i | s(x) > }` model the above described behavior of objects. Informally, one of these rules applies to states containing an object `< i | x >`, a message `credit(i)` for it, and (possibly) a subconfiguration `c1`.[3] If such a match is found, the state can be replaced by the term on the right-hand side of the rule (after applying the substitution of the match), resulting in a state without the message and where object `i` increments its balance with the successor operator `s`.

For the sake of simplicity, we assume that the system under study is described by a rewrite theory $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ whose rules are "topmost" for a designated kind `[State]` of states. We also assume that an operator {_} is used to enclose states so that all rules in R have that operator as their top operator in their left-hand sides. These assumptions are already quite general: they can cover, for example, object-based concurent systems. We further assume that $\mathcal{M}$ has *good executability properties*, i.e. that E is sufficient complete, (ground) confluent and terminating modulo A (that is, that the equational part correctly defines functions), and R is coherent with E modulo A [5] (that is, that applying equations to evaluate functions does not interfere with the application of the rules that specify system transitions). Moreover, unless we state the contrary, all extensions of $\mathcal{M}$ that we shall define will be required to be ground confluent, ground terminating, and sufficiently complete w.r.t. the same signature of constructors as $\mathcal{M}$. Fortunately, the standard Maude tools offer automatization support for checking such properties. Our running example satisfies all these conditions.

We consider the well-known semantic domain of Kripke structures for rewrite theories, suitable for state space exploration problems like model checking.

Definition 2 (Kripke structure). *A* Kripke structure K *is a tuple* $K = (S, \rightarrow, L, AP)$ *such that* S *is a set of states,* $\rightarrow \subseteq S \times S$ *is a transition relation between states, and* $L : S \rightarrow 2^{AP}$ *is a labelling function mapping states into sets of atomic propositions* AP *(i.e. observations on states).*

The Kripke semantics of a rewrite theory has `State`-sorted terms as states and one-step rewrites between `State`-sorted terms as transitions. The labelling function is defined by Boolean predicates specified equationally in the rewrite theory. As proved in [9], any computable Kripke structure, even an infinite-state one, can be obtained from an executable rewrite theory using only a finite signature Σ, and finite sets E of equations, A of axioms and R of rules.

[3] The second rule is needed since we treat the fact that `none` is an identity for union equationally rather than axiomatically.

Definition 3 (Kripke semantics of rewrite theories). *Let* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *be a rewrite theory with a designated state sort* `State`*, and a set* $AP \in \Sigma$ *of Boolean state predicates equationally defined in* E*. The Kripke structure associated to* $\mathcal{M}$ *is* $K_{\mathcal{M}} = (T_{State/E \cup A}, \rightarrow, L, AP)$ *such that* $T_{State/E \cup A}$ *are all* `State`*-sorted states,* $\rightarrow$ *is defined as* $\{[u] \rightarrow [v] \mid \mathcal{M} \vdash u \rightarrow^1_{R,State} v\}$ *(i.e. transitions are one-step rewrites between* $E \cup A$ *equivalence classes of* `State`*-terms in* $\mathcal{M}$*), and* L *is such that* $p \in L(s)$ *iff* $p(s) =_{E \cup A} true$*.*

We will consider bisimulation as the key semantic equivalence.

Definition 4 (bisimulation). *Let* $K = (S_K, \rightarrow_K, L_K, AP_K)$*,* $H = (S_H, \rightarrow_H, L_H, AP_H)$ *be two Kripke structures, and let* $\sim \subseteq S_K \times S_H$ *be a relation between* S_K *and* S_H*. We say that* $\sim$ *is a* bisimulation *between* K *and* H *iff for each two states* $s \in S_K$ *and* $s' \in S_H$ *such that* $s \sim s'$ *we have that: (i)* $L_K(s) = L_H(s')$*; (ii)* $s \rightarrow_K r$ *implies that there is a state* r' *s.t.* $s' \rightarrow_H r'$ *and* $r \sim r'$*; and (iii)* $s' \rightarrow_H r'$ *implies that there is a state* r *s.t.* $s \rightarrow_K r$ *and* $r \sim r'$*.*

The notion of bisimulation can be lifted to rewrite theories in the obvious way. We shall focus on bisimulations such that the relation $\sim$ is an equivalence relation, which includes the case of bisimulations induced by symmetries, i.e. when two states are bisimilar if they belong to the same class of symmetric states.

For instance, suppose that the initial state of our example is {`< 0 | 0 > < 1 | 0 > credit(0) credit(1)`}. We have then two possible transitions (given by the application of the rules governing the system), leading respectively to states: {`< 0 | 1 > < 1 | 0 > credit(1)`} and {`< 0 | 0 > < 1 | 1 > credit(0)`}. These two states are syntactically different but they are *symmetric*, i.e. equal up to the permutation of object identifiers.

Indeed, equivalence classes of symmetric states can be conveniently defined as the *orbits* of a group action (permutations in our example), which yield *symmetry reductions* as a special case of our approach. We hence recall here some basic notions about groups and group actions.

Definition 5 (group basics). *A* group *is a tuple* $G = (G, \bullet, e, (\_)^{-1})$ *where* G *is a set of elements,* $\bullet : G \times G \rightarrow G$ *is a binary associative operation,* $e \in G$ *is an identity (i.e.* $\forall f \in G. f \bullet e = e \bullet f = f$*), and* $(\_)^{-1}$ *is an inverse operator (i.e.* $\forall f \in G. f \bullet f^{-1} = f^{-1} \bullet f = e$*).*

Let G *be a group and* $H \subseteq G$ *be a subset of* G*. The group generated by* H *denoted* $\langle H \rangle$ *is defined as the closure of* H *under the inverse and product operators* $(\_)^{-1}$ *and* $\bullet$ *of* G*. In general* $\langle H \rangle$ *will be a subgroup of* G*, but if* $\langle H \rangle$ *coincides with* G*, then* H *is said to* generate G *and its elements are called* generators*.*

Let G *be a group and* A *be a set. An* action *of* G *on* A *is a monoid homomorphism* $[\![\cdot]\!] : G \rightarrow [A \rightarrow A]$*, that is,* $[\![f \bullet g]\!] = [\![f]\!] \circ [\![g]\!]$*, where* $f \circ g$ *denotes function composition in* $(A \rightarrow A)$*, and* $[\![e]\!] = id_A$*, with* id_A *the identity on* A*.*

Notable examples are permutation and rotation groups, which capture typical symmetries introduced by process replication in concurrent systems. Generators define groups in a concise manner, e.g. transpositions and single rotations for

permutation and rotation groups, respectively. The action of a group on the states of a Kripke structure implicitly defines an equivalence relation.

Definition 6 (equivalence induced by a group action). *Let S be a set of states, G be a group and $[\![\cdot]\!]$ be the action of G on S. Then the equivalence relation $\sim_G$ induced by G on S is defined by: $s \sim_G s' \Leftrightarrow \exists f \in G.[\![f]\!](s) = s'$.*

Group actions can be defined in rewriting logic with equations of the form `[[f]](t) = t'` where `f` denotes a group element (typically a generator) and `t`, `t'` are `State`-sorted terms. For instance, in our running example, the application of object identifier transpositions `i<->j` can be defined (by structural induction) with the equations:

```
 eq [teq1] : [[i<->j]]({c1}) = {[[i<->j]](c1)} .
 eq [teq2] : [[i<->j]](none) = none .
 eq [teq3] : [[i<->j]](c1 c2) = ([[i<->j]](c1)) ([[i<->j]](c2)) .
 eq [teq4] : [[i<->j]](< k | x >) = < [[i<->j]](k) | x > .
 eq [teq5] : [[i<->j]](credit(k)) = credit([[i<->j]](k) ) .
 eq [teq6] : [[i<->j]](i) = j .
ceq [teq7] : [[i<->j]](k) = k if (i != k) /\ (j != k) .
```

For example, the unconditional (`eq`) rule `teq 4` defines the application of a transposition `[[i<->j]]` to an `object (< k | x >)` as the object obtained by transposing its identifier. Equations `teq6` and `teq7` take care of transposing identifiers. A symmetric version of `teq6` is not needed since `_<->_` is commutative. Equation `teq7` is conditional (`ceq`): it applies when `teq6` is not applicable.

3 C-Reductions for Kripke Structures

We introduce the idea of *canonical reductions*, abbreviated c-reductions as a generic means to reduce a Kripke structure K by exploiting some equivalence relation $\sim$ on the states of K which is also a bisimulation on K (i.e. between K and itself). In Sect. 4 we will explain how c-reductions are specified, proved correct, and used for model checking in rewriting logic.

We start by defining canonizer functions, which are used to compute for a given state a (not necessarily unique) canonical representative of its equivalence class, modulo some equivalence relation which is also a bisimulation (e.g. a canonical permutation of the identifiers of processes with identical behavior).

Definition 7 (canonizer functions). *Let $K = (S, \rightarrow, L, AP)$ be a Kripke structure, and let $\sim \subseteq S \times S$ be a an equivalence relation which is a bisimulation on K. A function $\mathtt{c} : S \rightarrow S$ is a $\sim$-canonizer (resp. strong $\sim$-canonizer) iff for each $s \in S$ we have $s \sim \mathtt{c}(s)$ (resp. $s \sim \mathtt{c}(s)$, and $s \sim s' \rightarrow \mathtt{c}(s) = \mathtt{c}(s')$).*

Canonizer functions are used to compute smaller but semantically equivalent (i.e. bisimilar) Kripke structures by applying canonizers after each transition. Strong canonizers provide unique representatives for the equivalence classes of states and, hence, more drastic space reductions. That is, for two different but

equivalent states $s \sim s'$ they provide the same canonical representative (i.e. $\mathtt{c}(s) = \mathtt{c}(s')$). Typical examples of strong canonizers for equivalence classes are functions based on *enumeration* strategies [10] which generate the complete set of states of the equivalence class and then apply some function over it (e.g. based on a total ordering of the states). For instance, in our running example, an enumeration canonizer just generates all states that result from permuting (symmetric) processes in all possible ways and then selects one according to some total order (e.g. the lexicographic order of the description of states). In particular, for a state {`< 0 | 1 > < 1 | 0 > credit(1)`} the enumeration will produce its whole orbit: { {`< 0 | 1 > < 1 | 0 > credit(1)`}, {`< 0 | 0 > < 1 | 1 > credit(0)`}}. Then the canonizer would assign the least state of the set according to some total order, e.g. "identifier first, balance second" which would provide {`< 0 | 0 > < 1 | 1 > credit(0)`} as representative. Canonizers can be obtained in more efficient and smarter ways as shown in Sect. 4.4, e.g. with *local search strategies* [10] that repeatedly apply transpositions until the least state is reached. Instead, a non-strong (or weak) canonizer can provide different representatives for equivalent states. That is, it might be the case that $\mathtt{c}(s) \neq \mathtt{c}(s')$ even though $s \sim s'$. Weak canonizers provide weaker state space reductions, but they often enjoy advantages over strong canonizers: in some cases they are easier to be defined and analyzed, and their computation can be much more efficient in terms of runtime cost. Such heuristic canonizers can be found for instance in [11, 3], where the rough idea is to consider an ordering of the states that only depends on part of the state description. The resulting ordering relation is partial and the representative of a state is computed as one of the least states of the ordering.

The reduction of the state space is obtained by applying the canonizer to states after a transition. This is what we call a c*-reduction.*

Definition 8 (c-reduction of a Kripke structure). *Let $K = (S, \to, L, AP)$ be a Kripke structure, and let $\mathtt{c} : S \to S$ be a $\sim$-canonizer function for some equivalence relation $\sim \in S \times S$ which is a bisimulation on K. We call the Kripke structure $K/\mathtt{c} = (S, (\to; \mathtt{c}), L, AP)$ the* c-reduction *of K, where the composed transition relation $\to; \mathtt{c}$ is defined by $\to; \mathtt{c} = \{(s, \mathtt{c}(r)) \in S^2 \mid s \to r\}$.*

An important result is then that a c-reduction is bisimulation preserving.

Theorem 1 ($\sim$-preservation). *Let $K = (S, \to, L, AP)$ be a Kripke structure, let $\sim$ be an equivalence relation on S that is a bisimulation on K, and let $\mathtt{c}$ be a $\sim$-canonizer function. Then $\sim$ is a bisimulation relation between K and $K/\mathtt{c}$.*

4 Correct c-Reductions in Rewriting Logic

We now describe a methodology for specifying, proving correct, and analyzing c-reductions in rewriting logic. In this methodology, correctness proofs and model checking verification are supported by tools in the Maude formal environment such as the Maude LTL Model Checker [12], Invariant Analyzer [13], Inductive Theorem Prover [14] and Church Rosser and Coherence Checker [15].

We assume that there is some regularity in $\mathcal{M}$ that we try to exploit by defining an equivalence (bisimulation) relation $\sim$ on states to ease the analysis of $\mathcal{M}$. We also assume that the specification $\mathcal{M}$ satisfies the assumptions in Sect. 2 and is conveniently structured (see Fig. 1) into a core equational part ($\mathcal{M}.E$), and its extension with state predicate functions that define the atomic propositions ($\mathcal{M}.AP$) and behavioral rules ($\mathcal{M}.R$). Such modular structure is very natural and easy to achieve, and facilitates our methodology. Fig. 1 schematizes our methodology by identifying the main theories (or modules), their incremental construction via extensions (triple arrows) or refactoring (dashed arrows), and the modules involved in each step (dotted arrows). In particular, our methodology consists in the following steps: (i) specify and verify the equivalence relation $\sim$; when the equivalence $\sim_G$ is induced by a group G, specify the group action that induces $\sim_G$ in a module $\mathcal{M} + G$ and verify that it is indeed a group action (Sect. 4.1) which ensures $\sim_G$ to be an equivalence relation; (ii) verify that $\sim$ preserves the state predicates AP (Sect. 4.2) by analyzing their invariance under an auxiliary theory $\mathcal{M}/G$ that models group actions; (iii) verify that $\sim$ is a bisimulation (Sect. 4.3) by checking a coherence-like property between the rules of $\mathcal{M}.R$ and those of $\mathcal{M}/G$; (iv) define a canonizer c in a module $\mathcal{M} + \mathsf{c}$ and show it to be a $\sim$-canonizer (Sect. 4.4); (v) build the c-reduction $\mathcal{M}/\mathsf{c}$ of $\mathcal{M}$ (Sect. 4.5), and (vi) use $\mathcal{M}/\mathsf{c}$ for model checking analysis purposes. Our methology then ensures that any CTL* property φ holds on $\mathcal{M}/\mathsf{c}$ *if and only if* it holds on $\mathcal{M}$, since $\mathcal{M}/\mathsf{c}$ *has been proved to be a correct* c-reduction of $\mathcal{M}$, and therefore *bisimilar* to $\mathcal{M}$.

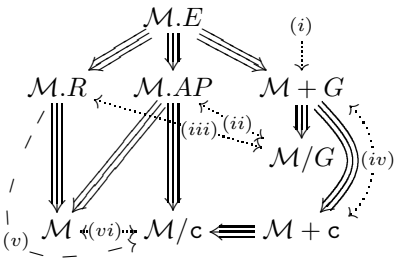


Fig. 1. Modules and steps

Some of the above steps are independent or apply at different levels of abstraction, so that they act as building blocks to be re-used as needed. For instance, verifying a c-reduction strategy does not require performing all the verification steps if it is based on a state equivalence that has been already proven to be correct. In practice, bisimulation relations and their canonizers need not be defined and proven correct for every system, as there will be classes of systems for which they can be specified once and for all. In such cases, one can define c-reductions as theory transformations for wide classes of examples corresponding, for instance, to certain permutation groups, or to other useful equivalence relations besides the symmetry reduction case. In Maude this can be done by exploiting reflection, so that the c-reduction is automatized as a function at the metalevel, possibly after checking some proof obligations. Our current prototype [7] applies some generic c-reductions to any object-based module.

Even though in some of the steps of our methodology we focus on c-reductions based on group actions, the c-reduction technique, in particular steps (v-vi), is more general and allows arbitrary canonizers. We focus on group actions to illustrate the ideas and the semi-automatic correctness checks (steps (i)-(iv))

with a simple example. More substantial examples can be found in [7]; several of them are mentioned, together with detailed performance experiments and comparisons with other tools and methods in [8, Sect. B].

4.1 Specifying and Verifying Group Actions

We give a simple method to equationally specify group actions and verify their correctness *in terms of a set H of generators only, without having to explicitly define the group G* generated by H. The key ideas, explained in detail in [8, Sect. E], consist on: (a) "uncurrying" the desired group action function $[\![\cdot]\!]_{\_} : G \longrightarrow (\texttt{State} \to \texttt{State})$ as a function $[\![\cdot]\!]_{\_} : G \times \texttt{State} \longrightarrow \texttt{State}$; (b) choosing a subset $H \subseteq G$ that generates G as a monoid; and (c) specifying the inverse $i(g) = g^{-1}$ of each generator $g \in H$ as a product of generators by a function $i : H \to H^*$, where H^* is the free monoid on the alphabet H.

The trick is that, after equationally specifying steps (b) and (c), *G needs not be explicitly defined*: it is enough to specify the action of the generators by a function $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$, which extends uniquely to a monoid action $[\![\cdot]\!]_{\_} : H^* \times \texttt{State} \to \texttt{State}$ satisfying for each $u \in \texttt{State}$, $g \in H$, $w \in H^*$ the recursive equations: $[\![\epsilon]\!]u = u$ and $[\![wg]\!]u = [\![w]\!]([\![g]\!]u)$. Then it is easy to prove (see [8, Sect. E]) that the only possible group action $[\![\cdot]\!]_{\_} : G \times \texttt{State} \to \texttt{State}$ extending $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$ exists if and only if the following equalities hold for each generator g and each state u: $[\![g]\!]([\![g^{-1}]\!](u)) = [\![g^{-1}]\!]([\![g]\!](u)) = u$. Then G needs not be explicitly specified, because we can safely replace G by the group $H^*/i = H^*/\{g \cdot i(g) = \epsilon \mid g \in H\}$, so that $\sim_G = \sim_{H^*/i}$, and the group action $[\![\cdot]\!]_{\_} : G \times \texttt{State} \to \texttt{State}$ can be replaced by the simpler monoid action $[\![\cdot]\!]_{\_} : H^* \times \texttt{State} \to \texttt{State}$.

The following definition captures (a) and (b), where we assume that H has been equationally specified by a new sort H, and then H^* has been specified by instantiating a parameterized module $List[X]$ to the instance $List[H]$.

Definition 9 (group pre-action specification). *Let $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ be the rewrite theory under study with designated $\texttt{State}$ sort. A* group pre-action *on $\mathcal{M}$ is an equational theory $\mathcal{M} + G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A, \emptyset, \phi)$ which is a protecting extension of the equational part of $\mathcal{M}$, $\mathcal{M}.E$, where Σ_G and E_G extend the equational theory $\mathcal{M}.E$ with a sort H, a sort H^* of lists of elements in H (i.e. the module $List[H]$ is protected in $\mathcal{M}+G$), a function $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$ recursively extended to a monoid action $[\![\cdot]\!]_{\_} : H^* \times \texttt{State} \to \texttt{State}$ as explained above, and a function $i : H \to H^*$.*

The *proof obligations* that need to be verified to show that a group pre-action is a group action are as follows:

Proposition 1 (correctness criteria for group actions). *Let $\mathcal{M} + G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A)$ be a group pre-action on $\mathcal{M}$. Then in the inital algebra of $\mathcal{M}+G$ the function $[\![\cdot]\!]_{\_} : H \times \texttt{State} \to \texttt{State}$ uniquely extends to a group action of H^*/i on $\texttt{State}$ if and only if the following two equations hold inductively in such an initial algebra: (i) $(\forall g : H, u : \texttt{State})\ [\![g]\!]([\![g^{-1}]\!](u)) = u$, and (ii) $(\forall g : H, u : \texttt{State})\ [\![g^{-1}]\!]([\![g]\!](u)) = u$.*

Using the above implicit definition method and checking the correctness criteria in the above proposition one can equationally define group actions and prove their correctness by inductive equational reasoning. In particular, this can be done for any group action of interest, defining symmetries between states, including the full and rotation symmetries that have been identified and thoroughly studied in the past. Note that sometimes (e.g. transpositions) $i(g) = g$, so that i needs not be defined explicitly, because it is the identity function. The action function $[\![\cdot]\!]\_ : H \times \texttt{State} \to \texttt{State}$ can be very easily specified in Maude, by topmost equations relating two `State`-sorted terms of the form `[[g]]({t}) = {t'}`, for (patterns of) elements $\texttt{g} \in H$.

Inductively showing that the equations (i) and (ii) in Proposition 1 are satisfied can usually be done easily by structural induction on the algebraic structure of states. For instance, to check that in our running example full symmetries yield a group action, all we have to do is to prove the equality $[\![i \leftrightarrow j]\!]([\![i \leftrightarrow j]\!](\{t\})) = \{t\}$, i.e. that applying the same transposition of i and j (denoted $i \leftrightarrow j$) twice amounts to applying the identity. This proof can be done by structural induction on `State`-sorted terms. For instance, to show that the property holds in the general case (i.e. `[[i<->j]] ([[i<->j]] ({c1 c2})) = {c1 c2}`), we apply the equations implementing the group action (namely `teq1`, `teq3`) to obtain `{[[i<->j]] ([[i<->j]] (c1)) [[i<->j]] ([[i<->j]] (c2))} = {c1 c2}` and conclude the proof by applying induction.[4]

Once proved that $\mathcal{M} + G$ correctly specifies a group action, we can conclude that the induced relation on states $\sim_G$ is actually an equivalence relation. In our example, we have an equivalence relation induced by object permutations.

4.2 Checking that $\sim$ Preserves Atomic Predicates

To prove that the equivalence $\sim_G$ induced by the action of group G preserves the atomic propositions AP we proceed as follows. First, we define a rewrite theory $\mathcal{M}/G$ for the sole purpose of analysis. The theory $\mathcal{M}/G$ is a protecting extension of $\mathcal{M} + G$ that introduces some rewrite rules to "move" inside orbits.

Definition 10. *Let $\mathcal{M}+G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A, \emptyset, \phi)$ be the theory specifying the action of a group G with generator $H \subseteq G$ on the states of a theory $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$. Then, the theory $\mathcal{M}/G$ is defined as $\mathcal{M}/G = (\Sigma \cup \Sigma_G, E \cup E_G \cup A, R_{\mathcal{M}/G}, \phi)$, where $R_{\mathcal{M}/G} = \{\texttt{\{t\} => [[g]]({t})} \mid \texttt{g} \in H\}$.*

In words, we replace the rules of $\mathcal{M}$ by rules that move from a state u to a state v obtained by applying a generator to u. If H is infinite, $R_{\mathcal{M}/G}$ is also infinite. However, in practice we can often find a finitary reformulation of $R_{\mathcal{M}/G}$, because $R_{\mathcal{M}/G}$ can often be expressed very concisely using patterns for the elements in H. For instance, the module `BANK/PERMUTATION` of our running example contains just two rules: `rl { < i | x > < j | y > } => { [[i<->j]] ( < i | x > < j | y >) }` and `rl { < i | x > < j | y > c1 } => { [[i<->j]] ( < i | x > < j | y > c1) }` to model transitions transposing two arbitrary objects.

[4] For the full proof see [8, Sect. C].

It is easy to see (by the properties of the generators of a group) that two states are reachable in $\mathcal{M}/G$ if and only if they are in the same orbit, i.e. that for any two states u, v we have the equivalence: $u \sim_G v \Leftrightarrow u \rightarrow^*_{R_{\mathcal{M}/G}} v$. Therefore, proving that a predicate $p \in AP$ is preserved by $\sim_G$, i.e. that for each pair of states $u, v \in T_{\mathtt{State}_{E/A}}$ $u \sim_G v$ implies $p(u) = p(v)$, is equivalent to proving that p is *stable* under $\mathcal{M}/G$, i.e. $\mathcal{M}/G \models (p \Rightarrow \Box p)$, where $\Box$ denotes the *always* operator of LTL.

To prove stability we need only to focus on the *positive* equations defining when p holds, which we assume are of the form `p({t}) = true`, or `p({t}) = true if cond`, with `cond` a condition. In our example, the predicate `some-message` characterizing states in which there is at least one message around for some existing object is defined by the equations `eq some-message( < i | x > credit(i)) = true` and `eq some-message( < i | x > credit(i) c1 ) = true`.

Under the assumptions that: (i) the constructors of $\mathcal{M}.E$ are *free modulo the axioms* A, and (ii) the terms `t` in predicate equations `p({t}) = true`, and the left-hand sides of rules in $\mathcal{M}/G$ are constructor terms, we can use the results in [16] to reduce proving $\mathcal{M}/G \models (p \Rightarrow \Box p)$ to the following proof obligations:

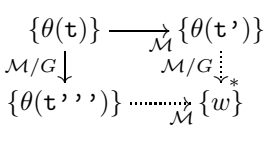
Proposition 2 (predicate preservation through stability). *Let $\mathcal{M}/G$ the auxiliary rewrite theory of Definition 10 and $\mathcal{M}$ satisfy assumptions (i)–(ii) above. and let* `p` *be an atomic proposition defined in $\mathcal{M}.AP$ by positive equations of the form described above. Then,* `p` *is preserved by $\sim_G$ iff for each rule* `{t'} => {t''}` $\in R_{\mathcal{M}/G}$*, each equation* `p({t}) = true` *in $\mathcal{M}.AP$, and each A-unifier ϑ[5], we can prove* `p({`ϑ`({t''})}) = true`*.*

Proposition 2 is very useful in practice, since we can use the Invariant Analyzer [13, 16] (InvA) to automate a good part of the effort of proving stability, leaving the remaining proof obligations for the Maude inductive theorem prover [14]. For example, the above mentioned proposition can be shown to be invariant under object permutations by InvA in a fully automatic way.

4.3 Checking that $\sim$ is a Bisimulation

Once the state relation $\sim$ we want to exploit has been shown to preserve the atomic propositions of interest, we have to check that $\sim$ is a bisimulation.

In the case of an equivalence relation $\sim_G$ induced by a group G, proving that $\sim_G$ is a bisimulation amounts to showing joinability of suitable "critical pairs" between the state transition rules `{t} => {t'}` in the rule set $\mathcal{M}$,[6] and the rules `{t''} => {t'''}` of $\mathcal{M}/G$.



Indeed, bisimulation is ensured if we prove that for all ground A-unifiers θ between `t` and `t''` and each corresponding critical pair denoted with ordinary arrows in the diagram on the right, there is a rule R giving us a one-step rewrite $\{\theta(\mathtt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$ for which we can prove: $\{\theta(\mathtt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$.

[5] Mappings of variables into non-necessarily ground terms such that $\vartheta(\mathtt{t'}) =_A \vartheta(\mathtt{t})$.

[6] The case of conditional rules in $\mathcal{M}$ is analogous, using *conditional* critical pairs.

Proposition 3 (correctness of bisimulation by joinability). *Let $\mathcal{M}$ be the rewrite theory under study, with an action of the group G. Then $\sim_G$ is a bisimulation between $\mathcal{M}$ and itself iff for all rules* {t} => {t'} *in $R_{\mathcal{M}}$, all rules* {t''} => {t'''} *in $\mathcal{M}/G$, and all ground A-unifiers θ between* t *and* t'' *there is a state $\{w\}$ such that $\{\theta(\texttt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$ and $\{\theta(\texttt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$.*

The above proposition requires considering the set of all ground A-unifiers which may be infinite. Fortunately, we can instead use A-unifiers with variables and, in particular, the *most general* ones. Since each ground A-unifier is an instance of a most general one, if we can prove the conditions in Proposition 3 for the finite set of most general A-unifiers, then we have proved bisimilarity. However, using the most general A-unifiers may not always automatically prove bisimilarity: some inductive joinability proof obligations may still be left.

That is, the use of most general A-unifiers yields the following *sound* and easy method to automate the proof. First, we use the Maude A-unification command to find the most general A-unifiers ϑ between {t} and {t''}, respectively the left-hand-sides of each rule {t} => {t'} of $\mathcal{M}$, and each rule {t''} => {t'''} of $\mathcal{M}/G$ (after a renaming of variables to ensure that they have no variables in common). Second, for each such A-unifier ϑ we can use the Maude search command to determine all possible 1-step rewrites $\{\vartheta(\texttt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$. Note that for each ground instance s of $\{\vartheta(\texttt{t'''})\}$ the obtained rewrite steps correspond to some of the possible transitions outgoing from state s. Last, we can use the search command again to check if at least one of such obtained terms $\{w\}$ can also be reached from $\{\vartheta(\texttt{t'})\}$ in $\mathcal{M}/G$. For example, applying this method to our running example yields six unifiers in the first step, each requiring one reachability check that is efficiently solved by the search command of Maude. Proposition 4 summarizes the method.

Proposition 4 (soundness of the bisimulation check). *Let $\mathcal{M}$ be the rewrite theory under study and $\sim_G$ an equivalence on states induced by the action of a group G. Then $\sim_G$ is a bisimulation between $\mathcal{M}$ and itself if for each rule* {t} => {t'} *in $\mathcal{M}$, rule* {t''} => {t'''} *in $\mathcal{M}/G$, and most general A-unifier ϑ between* t *and* t'', *there is one state $\{w\}$ with $\{\vartheta(\texttt{t'''})\} \rightarrow_{\mathcal{M}} \{w\}$ for which we can show $\{\vartheta(\texttt{t'})\} \rightarrow^*_{\mathcal{M}/G} \{w\}$.*

4.4 Defining and Verifying Canonizer Functions

The next step is to define canonizer functions c : State → State in a protecting extension $\mathcal{M}+\texttt{c}$ of the rewrite theory $\mathcal{M}$ under study. Note that in order to define c we may need to define some auxiliary functions (e.g. the ordering relations used in symmetry reduction to determine orbit representatives).

Definition 11 (c-extension of a rewrite theory). *Let $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ be the rewrite theory under study. A* c*-extension of $\mathcal{M}$ is a protecting extension of $\mathcal{M}$ of the form $\mathcal{M} + \texttt{c} = (\Sigma \cup \Sigma_{\texttt{c}}, E \cup E_{\texttt{c}} \cup A, R, \phi_{\texttt{c}})$ where $\texttt{c} \in \Sigma_{\texttt{c}}$ with* c : State → State, *and $\phi_{\texttt{c}}$ extends ϕ by making all functions in $\Sigma_{\texttt{c}}$ frozen.*[7]

[7] Imposing frozenness on the operators of $\Sigma_{\texttt{c}}$ is needed for the result of [8, Lemma 1].

Many candidate canonizers may exist for a given bisimulation $\sim$, each leading to different results in terms of the size of the reduced state space and computational performance. In any case, all canonizer functions must preserve $\sim$, i.e. they must be $\sim$-canonizers. This may require some theorem proving but it can be relatively easy to check in most cases, since we can use the equations E_c and show that each one preserves $\sim$.

For example, in the case of *local* reduction strategies [10] for symmetries based on a group G with generators $H \subseteq G$, the equations $E_{\mathtt{c}}$ defining `c` are of the form `c({t}) = c([[g]]({t})) if [[g]]({t})<{t}` with $g \in H$, `_<_` defining an ordering relation on states, plus an equation `c({t})={t} [owise]` to deal with the case when none of the previous equations is applicable, that is, when there is no way to transform a state into a *smaller* equivalent one by applying a generator (or inverse of a generator). Since such equations define `c` in terms of group actions or of the identity function when all conditions fail, preservation of the equivalence $\sim_G$ induced by G is immediate by the very definition of `c`.

Examples of local search strategies are implemented in our prototype tool [7]. In our running example, we can define such a canonizer as follows:

```
ceq   c( {            < i | x > < j | y > c1   } )
    = c( { [[i<->j]]( < i | x > < j | y > c1 ) } )
    if i < j /\ x < y .
eq c({c1}) = {c1} [ owise ] .
```

A very similar situation is that of *enumeration* strategies [10], where canonizers are defined as `c({t}) = min{[[f]]({t}) | f` $\in G$`}`. Again, preservation of $\sim_G$ by c follows from the very definition of `c`. Indeed, for all states u, `c`(u) will be necessarily of the form $[\![g_1 \bullet g_2 \bullet \cdots \bullet g_n]\!](u)$, with each g_i being a generator. We call the equation format described above *group application form*.

Proposition 5 (group application $\sim_G$ canonizers). *Let $\mathcal{M}$ be the rewrite theory under study, $\sim_G$ the state equivalence induced by a group action, $\mathcal{M}/G$ as in Definition 10, and $\mathcal{M}$ + `c` a `c`-extension of $\mathcal{M}$ such that the equations of $E_{\mathtt{c}}$ defining `c` are in* group application form*. Then, `c` is a $\sim_G$-canonizer.*

In practice, when specifying $\sim_G$-canonizers in the above form, all we have to check are the executability properties of the equations of $\mathcal{M}$ + `c`: termination, (ground) confluence and sufficient completeness, plus $\mathcal{M}$ protected in $\mathcal{M}$ + `c`, for which we can use the standard Maude tools.

Note that proving ground confluence of `c` is not sufficient to show that `c` is a *strong* canonizer. It may still be the case that for some two states u, v such that $u \sim v$ we have that `c`$(u) \neq$ `c`(v). For example, in the case of equivalences $\sim_G$ induced by a group G generated by $H \subseteq G$ as a monoid, to prove that `c` is a strong canonizer we also need to show that for all group elements in $g \in H$ and states s we have `c`$(s) =$ `c`$([\![g]\!](s))$. It is easy to see that if this holds, an inductive argument allows us to conclude `c`$(s) =$ `c`$([\![f]\!](s))$ for all $f \in G$ and hence for any two equivalent states $s \sim_G s' = [\![f]\!](s)$. Of course, there are cases in which no check is needed. For instance, it is well-known that enumeration strategies yield strong canonizers, while local strategies are not strong in general.

4.5 Defining c-Reductions

The next step is defining a c-reduction of $\mathcal{M}$ as a rewrite theory $\mathcal{M}/\mathtt{c}$. This is very useful, since then *no changes to a model checker are needed to support c-reductions*: we just model check $\mathcal{M}/\mathtt{c}$. We show that $\mathcal{M}/\mathtt{c}$ can be easily obtained by applying a theory transformation $\mathcal{M} \mapsto \mathcal{M}/\mathtt{c}$ defined as follows.

Definition 12 (c-reduction of a rewrite theory). *Let* $\mathcal{M} + \mathtt{c} = (\Sigma \cup \Sigma_\mathtt{c}, E \cup E_\mathtt{c} \cup A, R, \phi_\mathtt{c})$ *be a* c*-extension of* $\mathcal{M} = (\Sigma, E \cup A, R, \phi)$ *for which* c *is a* $\sim$*-canonizer of an equivalence bisimulation* $\sim$*. We then call* $\mathcal{M}/\mathtt{c} = (\Sigma \cup \Sigma_\mathtt{c}, E \cup E_\mathtt{c} \cup A, R_\mathtt{c}, \phi_\mathtt{c})$ *a* c-reduction *of* $\mathcal{M}$*, where* $R_\mathtt{c} = \{\mathtt{t => c(t') if cond} \mid (\mathtt{t => t' if cond}) \in R\}$.

$\mathcal{M}/\mathtt{c}$ is very much like $\mathcal{M}$, except that each rule `t => t' if cond` in R, is transformed into a rule `t => c(t') if cond`, i.e. into a rule where the canonizer function `c` is applied to the right hand side to ensure that canonization is performed after each system transition. For our running example we obtain, e.g. a rule: `rl { < i | x > credit(i) c1 } => c({ < i | s(x) > c1 }).`

This transformation is supported by our prototype [7] for the class of object-based rewrite theories. Our transformation exploits Maude reflective features: it is defined by a function that manipulates the meta-representation of the input theory to be c-reduced.

For some rules in $\mathcal{M}/\mathtt{c}$ it may be more efficient not to apply the canonizer after each step. For instance, if we know that the corresponding rule in $\mathcal{M}$ will always result in a canonical state we can save the time of applying the canonizer.

It is trivial to show that $\mathcal{M}/\mathtt{c}$ is a c-reduction by construction, and in particular that $\mathcal{K}_\mathcal{M}/\mathtt{c} = \mathcal{K}_{\mathcal{M}/\mathtt{c}}$. It can also be shown that it has good executability properties. By the properties required for $E_\mathtt{c}$, it inherits all the properties of the equational part of $\mathcal{M}$, namely sufficient completeness, confluence and termination modulo A. Moreover, it can be shown that $\mathcal{M}/\mathtt{c}$ is coherent modulo A.

Theorem 2 (executability of $\mathcal{M}/\mathtt{c}$). *Let* $\mathcal{M}$ *be the rewrite theory under study, and let* $\mathcal{M}/\mathtt{c}$ *be as in Definition 12. If* $\mathcal{M} + \mathtt{c}$ *has good executability properties, then* $\mathcal{M}/\mathtt{c}$ *also has good executability properties.*

The above theorem means that we can use $\mathcal{M}/\mathtt{c}$ for model checking analysis. For example, in our running example we can use the Maude LTL model checker to successfully verify the property $\Diamond\Box\neg$`some-message` (*"eventually there will be no more messages forever"*) efficiently. If we explore the whole state space of our running example using the Maude reachability analyzer we can check that the state space of the c-reduced system is drastically smaller than that of the original system. For instance, if we choose an initial state with 4 empty accounts with 4 messages for each, the original state space has 625 states, while the c-reduced one has only 70.

This is just a simple example: the performance experiments reported in [8, Sect. B] include examples taken from the literature where the applied c-reductions provide drastic gains and allow analyzing systems whose original state spaces is too large to be effectively analyzed.

5 Related Work and Conclusions

Related Work. We briefly comment on some interesting related approaches besides the ones already mentioned. A complementary line of research focuses on automatic symmetry detection, proposed for some model checkers, e.g. SPIN [10] and ProB [17]. Our approach does not forbid (though does not yet provide) automatically detected symmetries but focuses on user-definable ones, providing a methodology to check their correctness, with the main advantage being that we rely on tools and techniques used to perform the verification of the system itself. A related work is reported in [18] where formal methods are used to prove the soundness of the reduction techniques of [17].

Interesting are as well other state space reduction techniques, in particular those already proposed in the setting of rewriting logic and Maude, such as partial order reduction [19], and equational abstraction [20]. The closest one is [20], where abstractions are defined equationally. The main difference with our approach is in the kind of behavioral equivalence considered: equational abstractions yield simulations while we focus on bisimulations. With respect to [19] our approach is orthogonal and we are hence investigating how to combine them to improve the efficiency of rewriting-logic based interpreters of programming languages, in particular those with primitives for dynamic memory allocation.

Conclusions. We have presented *c-reductions*, a general bisimulation-based reduction technique that exploits canonizer functions whenever a bisimulation is an equivalence relation. The main differentiating features with respect to other state space reduction techniques are: (i) no changes to the underlying model checker are required, and reductions are defined using the original system description language; (ii) model checking and correctness proofs for the reduction are seamlessly integrated and supported by tools; (iii) semi-automation: both for applying the reduction and for checking their correctness; and (iv) generality: it subsumes in a uniform way symmetry reduction as well as other kinds of reductions (e.g. name reuse and name abstraction).

We have presented the basic concepts, described some typical classes of reductions, and illustrated how they can be analyzed. Our methodology performs a series of incremental steps Sect. 4.1–4.5, which include checking that the equivalence relation is a bisimulation and that the reduction strategies preserve such equivalence relation. Even if not presented here, we have performed a set of experimental results (see [8, Sect. B]) were we have observed a comparable performance with respect to symmetry reduction extensions of mature tools such as SPIN and performance gains with respect to previous implementations of symmetry reduction in Maude [6].

The flexibility of our approach has allowed us to define a wide range of reductions. Beyond the classical permutation and rotation symmetries, we have considered some simple cases of name reuse and name abstraction, which are crucial to deal with the infinite state spaces of systems with dynamic allocation of resources. Indeed, compared to the approach presented in [3, 11] we are able to treat a wider class of systems, where identifiers of symmetric objects can appear

as pointers in attributes of other objects, and with wider classes of symmetries such as rotational ones. Similar remarks can be made about [6], with respect to which we offer a wider class of reduction strategies and better performance.

Even though we have emphasized reductions based on group actions, the c-reduction approach is more general and accepts any possible canonizer function. Correctness proof methods fully covering the general case should be developed in future work. A preliminary version of our tool is publicly available [7].

References

1. Wahl, T., Donaldson, A.F.: Replication and abstraction: Symmetry in automated formal verification. Symmetry 2, 799–847 (2010)
2. Hendriks, M., Behrmann, G., Larsen, K., Niebert, P., Vaandrager, F.: Adding Symmetry Reduction to Uppaal. In: Larsen, K.G., Niebert, P. (eds.) FORMATS 2003. LNCS, vol. 2791, pp. 46–59. Springer, Heidelberg (2004)
3. Bosnacki, D., Dams, D., Holenderski, L.: Symmetric SPIN. International Journal on Software Tools for Technology Transfer 4, 92–106 (2002)
4. Meseguer, J.: Conditional rewriting logic as a united model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Rodríguez, D.E.: Combining techniques to reduce state space and prove strong properties. In: WRLA. ENTCS, vol. 238(3), pp. 267–280 (2009)
7. C-Reducer, `http://sysma.lab.imtlucca.it/tools/c-reducer`
8. Lluch Lafuente, A., Meseguer, J., Vandin, A.: State space c-reductions of concurrent systems in rewriting logic (2012), Full version, `eprints.imtlucca.it/1350`
9. Meseguer, J., Palomino, M., Martí-Oliet, N.: Algebraic simulations. Journal of Logic and Algebraic Programming 79, 103–143 (2010)
10. Donaldson, A.F., Miller, A.: A Computational Group Theoretic Symmetry Reduction Package for the Spin Model Checker. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 374–380. Springer, Heidelberg (2006)
11. Bošnački, D., Dams, D., Holenderski, L.: A Heuristic for Symmetry Reductions with Scalarsets. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 518–533. Springer, Heidelberg (2001)
12. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL Model Checker and Its Implementation. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 230–234. Springer, Heidelberg (2003)
13. The Maude Invariant Analyzer Tool (InvA), `http://camilorocha.info/software/inva`
14. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. Journal of Universal Computer Science 12, 1618–1650 (2006)
15. Durán, F., Meseguer, J.: A Church-Rosser Checker Tool for Conditional Order-Sorted Equational Maude Specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010)
16. Rocha, C., Meseguer, J.: Proving Safety Properties of Rewrite Theories. In: Corradini, A., Klin, B., Cîrstea, C. (eds.) CALCO 2011. LNCS, vol. 6859, pp. 314–328. Springer, Heidelberg (2011)
17. Spermann, C., Leuschel, M.: ProB gets nauty: Effective symmetry reduction for B and Z models. In: TASE, pp. 15–22. IEEE Computer Society (2008)

18. Turner, E., Butler, M., Leuschel, M.: A Refinement-Based Correctness Proof of Symmetry Reduced Model Checking. In: Frappier, M., Glässer, U., Khurshid, S., Laleau, R., Reeves, S. (eds.) ABZ 2010. LNCS, vol. 5977, pp. 231–244. Springer, Heidelberg (2010)
19. Farzan, A., Meseguer, J.: Partial order reduction for rewriting semantics of programming languages. In: WRLA. ENTCS, vol. 176(4), pp. 61–78 (2007)
20. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. Theoretical Computer Science 403, 239–264 (2008)

A Practical Loop Invariant Generation Approach Based on Random Testing, Constraint Solving and Verification

Mengjun Li*

School of Computer, National University of Defense Technology,
Changsha, China
mengjun_li1975@yahoo.com.cn

Abstract. Loop invariants play a major role in software verification. Based on random testing, constraint solving and verification, this paper presents a practical approach for generating equality loop invariants. More importantly, we present a practical verification approach of loop invariant based on finite difference technique. This approach is efficient since the constraint system is linear equational system. The effectiveness of the approach is demonstrated on examples.

Keywords: Loop Invariant Generation, Random Testing, Finite Difference.

1 Introduction

A key problem in automatic software verification system is the inference of loop invariants. A loop invariant is an assertion that is valid before and after each loop iteration. Dynamic approach provides ways to discover likely invariants rapidly. In a nutshell, the dynamic approach consists in testing candidate invariant templates against several program runs, the invariant templates that are not violated in any of the executions are retained as likely invariants. In dynamic approach, these provided invariant templates are all linear.

In this paper, we present a practical approach for generating equality loop invariants using random testing, constraint solving and verification. Given a template of equality loop invariants, by sufficiently random testing the given loop, a linear equation constraints system on the unknown coefficients in the template is established, by solving the equation system, a likely loop invariant is generated, and its validity is verified by computing its finite differences over all program transitions.

In our approach, the likely loop invariants is discovered firstly and their validity is verified secondly, these two steps are disconnected. With this strategy the constraints system generated is a linear equation system, they can be solved

* This work was supported by the National Natural Science Foundation of China (Grant Nos. 60703075, 90718017).

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 447–461, 2012.

```
n=0;
f=1;
while(f≤N)
{
  n=n+1;
  f=n*f;
}
```

Fig. 1. Program factorial.c

efficiently. Our approach differs from other template-based related work, where discovering loop invariants and verifying their validity are integrated, and non-linear constraints system is generated. Since many template-based techniques require solving non-linear constraints, their applicability is limited by the lack of efficient algorithms for solving such constraints.

More importantly, we present a practical verification approach of equality loop invariant based on finite difference technique. With this verification approach, we can prove $f - n! = 0$ is a loop invariant of the program factorial.c in Figure.1. Note that $f - n! = 0$ is not a polynomial loop invariant, to the best of our knowledge, our work is the first to generate non-polynomial loop invariant.

The rest of this paper is structured as follows: section 2 presents some preliminary definitions, section 3 introduces our approach on a typical example, section 4 presents a practical verification approach of equality loop invariant based on its finite difference over all program transitions and presents the algorithm for automated generating loop invariants, the results of our experiments are shown in section 5, section 6 discusses the related work, and section 7 concludes this paper.

2 Preliminaries

Let $x_1, \cdots, x_n$ be program variables, and let $\overline{x} = (x_1, \cdots, x_n)$.

Definition 1. *(Assignment Transition)An assignment transition τ is an assignment of the form $\overline{x} := (f_1(\overline{x}), \cdots, f_n(\overline{x}))$, where $\overline{x} = (x_1, \cdots, x_n)$, and $f_1, \cdots, f_n$ are n functions over program variables $x_1, \cdots, x_n$.*

An assignment transition $\tau : \overline{x} := (f_1(\overline{x}), \cdots, f_n(\overline{x}))$ can be rewritten to an assertion $x_1' = f_1(\overline{x}), \cdots, x_n' = f_n(\overline{x})$, where x_i and $x_i'(1 \leq i \leq n)$ are the pre- and post- variables of the transition τ, respectively.

In this paper, we focus on the multipath program defined as follows:

Definition 2. *(Multipath Program)For variables $\overline{x} = (x_1, \cdots, x_n)$, a multipath program with m paths has the form shown in Figure.2, where $\overline{x_0}$ expresses the initial value of $\overline{x}$, and $f_i^j(1 \leq i \leq n, 1 \leq j \leq m)$ are functions over program variables $x_1, \cdots, x_n$.*

$\overline{x} := \overline{x_0}$
while *true* **do**
  $\tau_1 : \overline{x} := (f_1^1(\overline{x}), \cdots, f_n^1(\overline{x}))$
    or
    $\vdots$
    or
  $\tau_m : \overline{x} := (f_1^m(\overline{x}), \cdots, f_n^m(\overline{x}))$
od

Fig. 2. multipath program

The program model in [7] and [8] is also the multipath program.

Definition 3. *(Loop Invariant). For a multipath program shown in Figure.2, a loop invariant is an assertation $E(\overline{x}) = 0$ that satisfies the following conditions:*

(1) initial condition : $E(\overline{x_0}) = 0$
(2) iterative condition : the Hoare triple $\{E(\overline{x}) = 0\}(\tau_1|\cdots|\tau_m)^*\{E(\overline{x}) = 0\}$ holds.

where $|$ denotes the alternation operator, and $*$ denotes the kleene closure operator in regular expression theory.

In the above definition, $\tau_1|\cdots|\tau_m$ denotes non-deterministic choice between $\tau_1, \cdots, \tau_m$, and $(\tau_1|\cdots|\tau_m)^*$ denotes the set of finite words over the alphabet $\Sigma = \{\tau_1, \cdots, \tau_m\}$. For convenience, the notation $\{E(\overline{x}) = 0\}(\tau_1|\cdots|\tau_m)^*\{E(\overline{x}) = 0\}$ is abused, which denote that $\{E(\overline{x}) = 0\}w\{E(\overline{x}) = 0\}$ holds for each word w over the alphabet $\Sigma = \{\tau_1, \cdots, \tau_m\}$.

Intuitively, $E(\overline{x}) = 0$ is a loop invariant if and only if the conditions below are satisfied:

(1) $E(\overline{x}) = 0$ is valid for the initial values of the loop variables. Namely, $E(\overline{x})$ is 0 whenever $\overline{x}$ are evaluated at $\overline{x_0}$. Thus, $E(\overline{x}) = 0$ holds at the entry of the loop.
(2) $E(\overline{x}) = 0$ is valid after arbitrary many execution of $\tau_1, \cdots, \tau_m$ in any order, starting in a state in which the initial values of the loop variable $\overline{x}$ are $\overline{x_0}$. Thus, $E(\overline{x}) = 0$ is preserved by any execution of the multipath program.

Definition 4. *(Finite Difference). The finite difference of a likely loop invariant $E(\overline{x}) = 0$ over an assignment transition τ is the expression $E(\overline{x'}) - E(\overline{x})$, denoted by $\Delta_\tau E(\overline{x})$, where τ gives the value of $\overline{x'}$ in terms of $\overline{x}$.*

Definition 5. *The finite difference tree (FDT) of a likely loop invariant $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ is defined as follows:*

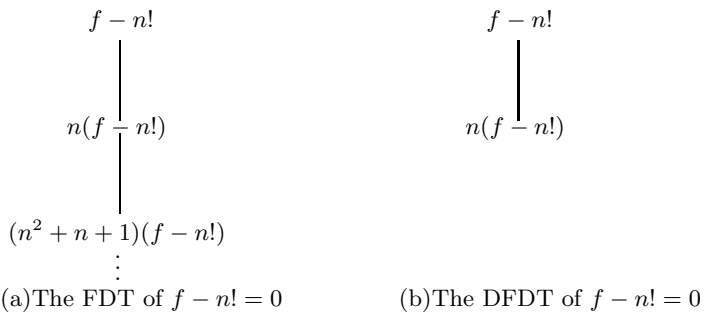
(1) The root is $E(\overline{x})$ and the leaves are values 0;
(2) If the tree contains a non-leaf node $F(\overline{x})$, then $F(\overline{x})$ has m child nodes $\Delta_{\tau_1} F(\overline{x}), \cdots, \Delta_{\tau_m} F(\overline{x})$.

Definition 6. *Let T be a finite difference tree of a likely loop invariant $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$, a node $F(\overline{x})$ in T is called a zero node if $F(\overline{x}) = \sum_{j=0}^{k} p_j(\overline{x})F_j(\overline{x})$, where $F_j(\overline{x})(j = 0, \cdots, k))$ is the ancestor node of $F(\overline{x})$ and each $F_j(\overline{x})$ satisfies that $F_j(\overline{x_0}) = 0(j = 0, \cdots, k)$, each $p_j(\overline{x})(j = 0, \cdots, k)$ is an arbitrary function over variables $x_1, \cdots, x_n$.*

Definition 7. *The decidable finite difference tree(DFDT) of a likely loop invariant $E(x) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ is the finite difference tree of $E(x)$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ with zero nodes as leaves.*

The height of a DFDT is the longest path to a leaf. A finite DFDT is a decidable finite difference tree with finite height.

For the program factorial.c in Figure.1, the FDT the DFDT of the likely loop invariant $f - n! = 0$ with respect to the transition $\tau : (n, f) = (n + 1, f * n)$ are presented in Figure.3. Note that $n(f - n!)$ is a zero node, although the FDT is infinite, the DFDT is finite.



(a)The FDT of $f - n! = 0$ (b)The DFDT of $f - n! = 0$

Fig. 3. The FDT and the DFDT of $f - n! = 0$

3 Motivation Example

We illustrate our approach using the example program cubicroot.c shown in Figure.4.

The imperative loop implements an algorithm for computing the cubic root r for a given integer number a, we guess a loop invariant template $c_1 + c_2a + c_3r + c_4r^2 + c_5r^3 + c_6x + c_7s = 0$, where $c_i(i = 1, \cdots, 7)$ are unknown coefficients. Then how to decide whether the above loop has a loop invariant in this form or not, and how to compute the values of the unknown coefficients $c_i(i = 1, \cdots, 7)$?

By the definition of loop invariant, all the values of a, r, x, s at the exit of the loop satisfy the template, for instance, if we have $a = 0$ at the entry of the loop, then we have $a = 0, x = 0, r = 1, s = \frac{13}{4}$ at the exit, since the values of a, r, x, s at the exit satisfy the template $c_1 + c_2a + c_3r + c_4r^2 + c_5r^3 + c_6x + c_7s = 0$, we have $c_1 + c_3 + c_4 + c_5 + \frac{13}{4}c_7 = 0$.

```
x=a;
r=1;
s=13/4;
while(x-s>0)
{
  x=x-s;
  s=s+6*r+3;
  r=r+1;
}
```

Fig. 4. Program cubicroot.c

Sampling m copies $a_i, r_i, x_i, s_i (i = 1, \cdots, m)$ of values of a, r, x, s, a linear equation system as follows will be obtained:

$$\begin{pmatrix} 1 & a_1 & r_1 & r_1^2 & r_1^3 & x_1 & s_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & a_m & r_m & r_m^2 & r_m^3 & x_m & s_m \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

With the knowledge of linear equation system, among all the vectors $\langle 1, a_k, r_k, r_k^2, r_k^3, x_k, s_k \rangle (k = 1, \cdots, m)$, if there exist $l (l > 7)$ vectors $\langle 1, a_{i_j}, r_{i_j}, r_{i_j}^2, r_{i_j}^3, x_{i_j}, s_{i_j} \rangle$ $(j = 1, \cdots, l)$ are linear independent, the linear equation system has no solutions. If there exist seven vectors $\langle 1, a_{i_j}, r_{i_j}, r_{i_j}^2, r_{i_j}^3, x_{i_j}, s_{i_j} \rangle (j = 1, \cdots, 7)$ are linear independent, then the solutions of all the unknown coefficients $c_i (i = 1, \cdots, 7)$ are 0. And if there exist $l (1 \leq l < 7)$ vectors $\langle 1, a_{i_j}, r_{i_j}, r_{i_j}^2, r_{i_j}^3, x_{i_j}, s_{i_j} \rangle (j = 1, \cdots, l)$ are linear independent, then the solutions of the linear equation system are linear combination of $n - l$ linear independent vectors.

The above observation indicates that if there exist $l (l \geq 7)$ linear independent vectors $\langle 1, a_{i_j}, r_{i_j}, r_{i_j}^2, r_{i_j}^3, x_{i_j}, s_{i_j} \rangle (j = 1, \cdots, l)$ among all the vectors $\langle 1, a_k, r_k, r_k^2, r_k^3, x_k, s_k \rangle (k = 1, \cdots, m)$, the above loop has no loop invariant in the form $c_1 + c_2 a + c_3 r + c_4 r^2 + c_5 r^3 + c_6 x + c_7 s = 0$. If there exist $l (1 \leq l < 7)$ vectors $\langle 1, a_{i_j}, r_{i_j}, r_{i_j}^2, r_{i_j}^3, x_{i_j}, s_{i_j} \rangle (j = 1, \cdots, l)$ are linear independent, then the linear equation system has solutions, replace the unknown coefficients $c_i (i = 1, \cdots, 6)$ in the template $c_1 + c_2 a + c_3 r + c_4 r^2 + c_5 r^3 + c_6 x + c_7 s = 0$ with their solutions, we obtain a likely loop invariant. Since not all values of a, r, x, s are sampled, the likely loop invariant may not be a real loop invariant.

We use random testing technique to automatically sample values of a, r, x, s. In random testing technique, the pseudo random numbers are generated as inputs of programs. For the program cubicroot.c, we utilize the Mathematica program cubicroot.m in Figure.5. to random testing and discover likely loop invariants .

In cubicroot.m, the variable *sample* denotes the number of the generated pseudo random inputs, and the variable *counter* denotes the sampling

number of values of program variables, $array1, array2, array3$ and $array4$ are arrays recording the values of a, x, s, r respectively, $RandomInteger[\{i_{min}, i_{max}\}]$ is a function used to generate a pseudo-random integer number in the range $[i_{min}, i_{max}]$, $Table[expr, \{i, i_{max}\}]$ is a function used to generate a list of values of $expr$ when i runs from 1 to i_{max}, $Solve[eqns, vars]$ is a function used to solve an equation or a set of equations for the variables $vars$.

In Table 1, the solutions of the linear equation system induced by different samples are listed. From Table 1, we find the solutions are all equal when the sampling number $counter \geq 5$:

$$C_1 = -\frac{1}{4}C_6 - \frac{1}{4}C_7, \quad C_2 = -C_6, \quad C_3 = \frac{3}{4}C_6$$
$$C_4 = -\frac{3}{2}C_6 - 3C_7, \qquad C_5 = C_6$$

, where C_6, C_7 are free variables. Replace the unknown coefficients $c_i (i = 1, \cdots, 7)$ in the template $c_1 + c_2 a + c_3 r + c_4 r^2 + c_5 r^3 + c_6 x + c_7 s = 0$ with the stable solution, we obtain the likely loop invariant $C_6(-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x) + C_7(-\frac{1}{4} - 3r^2 + s) = 0$. Note that C_6, C_7 are free variables, then $-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x = 0$ and $-\frac{1}{4} - 3r^2 + s = 0$ are both likely loop invariants.

Since the likely loop invariant may not be a real loop invariant, further we

Table 1. Solutions

Counter	Solutions
4	$C_1 = -\frac{149328}{263}C_5 + \frac{597049}{1052}C_6 - \frac{1}{4}C_7, C_2 = -\frac{6}{263}C_5 - \frac{257}{263}C_6,$ $C_3 = -\frac{54307}{263}C_5 - \frac{216439}{1052}C_6, C_4 = -\frac{6510}{263}C_5 + \frac{12231}{526}C_6 - 3C_7$
5	$C_1 = -\frac{1}{4}C_6 - \frac{1}{4}C_7, C_2 = -C_6, C_3 = \frac{3}{4}C_6,$ $C_4 = -\frac{3}{2}C_6 - 3C_7, C_5 = C_6$
6	$C_1 = -\frac{1}{4}C_6 - \frac{1}{4}C_7, C_2 = -C_6, C_3 = \frac{3}{4}C_6,$ $C_4 = -\frac{3}{2}C_6 - 3C_7, C_5 = C_6$
50	$C_1 = -\frac{1}{4}C_6 - \frac{1}{4}C_7, C_2 = -C_6, C_3 = \frac{3}{4}C_6,$ $C_4 = -\frac{3}{2}C_6 - 3C_7, C_5 = C_6$
100	$C_1 = -\frac{1}{4}C_6 - \frac{1}{4}C_7, C_2 = -C_6, C_3 = \frac{3}{4}C_6,$ $C_4 = -\frac{3}{2}C_6 - 3C_7, C_5 = C_6$

verify the validity of the likely loop invariants $-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x = 0$ and $\frac{1}{4} + 3r^2 - s = 0$. By theorem 1 in section 4, to be real loop invariants, the likely loop invariants $-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x = 0$ and $\frac{1}{4} + 3r^2 - s = 0$ should have finite DFDTs.

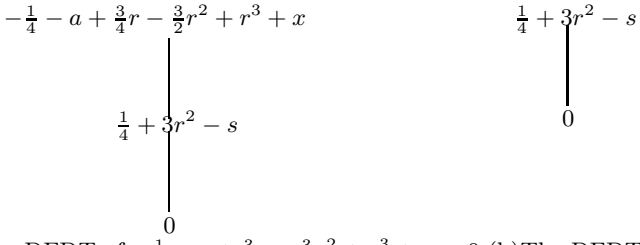
The finite DFDTs of $-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x = 0$ and $\frac{1}{4} + 3r^2 - s = 0$ are presented in Figure 6. By theorem 1, both $-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x = 0$ and $\frac{1}{4} + 3r^2 - s = 0$ are loop invariants.

4 Automated Generation of Loop Invariants

Based on the DFDT, the following theorem presents a practical verification approach for verifying the validity of a likely loop invariant.

```
sample=5;
counter=5;
array1=Array[aValue,counter];
array2=Array[xValue,counter];
array3=Array[sValue,counter];
array4=Array[rValue,counter];
l=1; n=1;
While[n≤sample,
  a=RandomInteger[{1, 1000}];
  x=a;
  r=1;
  s=13/4;
  While[x-s>0,
    x=x-s;
    s=s+6*r+3;
    r=r+1;
    If[l≤counter,
       Part[array1,l]=a;
       Part[array2,l]=x;
       Part[array3,l]=s;
       Part[array4,l]=r;
       l=l+1,
       Break[];];];
    If[l>counter,Break[];];
    n=n+1];
m=Table[C1+C2*Part[array1,j]+C3*Part[array4,j]+C4*Part[array4,j]^2
+C5*Part[array4,j]^3+C6*Part[array2,j]+C7*Part[array3,j]==0,j,counter];
solutions=Solve[m,{C1,C2,C3,C4,C5,C6,C7}];
```

Fig. 5. Program cubicroot.m



(a)The DFDT of $-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x = 0$ (b)The DFDT of $\frac{1}{4} + 3r^2 - s = 0$

Fig. 6. The DFDTs of the program cubicroot.c

Theorem 1. *For a multipath program shown in Figure.2, if there exists a finite DFDT T of $E(\overline{x}) = 0$ with respect to transitions $\mathcal{T} = \{\tau_1, \cdots, \tau_m\}$ and $E(\overline{x_0}) = 0$, then $E(\overline{x}) = 0$ is a loop invariant.*

Proof. We simultaneously prove that: for each node $F(\overline{x})$ in T, $F(\overline{x}) = 0$ is a loop invariant. We first prove that for a non-leaf node $F(\overline{x})$, $F(\overline{x}) = 0$ is a loop invariant.

We prove the initial condition $F(\overline{x_0}) = 0$ holds firstly. If $F(\overline{x})$ is not a leaf, then it is an ancestor node of some leaf, then $F(\overline{x_0}) = 0$.

Secondly, we assume that $F(\overline{x}) = 0$ holds after $n(n \geq 0)$ times execution of $\tau_1, \cdots, \tau_m$ in any order, and we prove that $F(\overline{x}) = 0$ holds after $(n+1)(n \geq 0)$ times execution of $\tau_1, \cdots, \tau_m$ in any order. If $F(\overline{x})$ is not a leaf and τ_i is executed, then $F(\overline{x'}) - F(\overline{x}) = G(\overline{x})$, where $G(\overline{x})$ is the child of $F(\overline{x})$. If $G(\overline{x})$ is not a leaf, since $G(\overline{x}) = 0$ and $F(\overline{x}) = 0$ by the assumption, then $F(\overline{x'}) = 0$. If $G(\overline{x})$ is a leaf, then $G(\overline{x})$ is a zero node, $G(\overline{x}) = \sum_{j=0}^{k} p_j(\overline{x}) F_j(\overline{x})$, where $F_j(\overline{x})(j = 0, \cdots, k))$ is the ancestor node of $G(\overline{x})$, by the assumption, both $F(\overline{x}) = 0$ and $F_j(\overline{x}) = 0(j = 0, \cdots, k)$ hold, then $F(\overline{x'}) = 0$ holds. In both cases, the iterative condition $\{F(\overline{x}) = 0\}\tau_i\{F(\overline{x}) = 0\}$ holds.

We conclude that: for each non-leaf node $F(\overline{x})$, $F(\overline{x}) = 0$ is a loop invariant.

If $F(\overline{x})$ is a leaf, then $F(\overline{x}) = \sum_{j=0}^{k} p_j(\overline{x}) F_j(\overline{x})$, where $F_j(\overline{x})(j = 0, \cdots, k))$ is the ancestor node of $F(\overline{x})$, since each $F_j(\overline{x}) = 0$ is a loop invariant, then $F(\overline{x}) = 0$ is a loop invariant.

Since $E(\overline{x})$ is the root of T, particularly, $E(\overline{x}) = 0$ is a loop invariant. □

By theorem 1 and the DFDT in Figure.3, $f - n! = 0$ is a loop invariant of the program factorial.c.

The algorithm for automated generation of loop invariants is described in Figure.7. In Figure.7, the function $randomSampling(P, varList, n)$ denotes the multipath program P is random tested n times, and the sampled values of program variables in $varList$ are stored in the array $valueArray$, the function $generatingLinearSystem(valueArray, \eta)$ denotes the linear equation constraint system Ψ is established with the sampling values $valueArray$ and the template η, the function $solvingLinearSystem(\Psi)$ denotes the linear equation constraint system Ψ is solved and the solution is $\mathcal{P}^*$, if the solution is 0(denoted by $\mathcal{P}^* = 0$) or Ψ has no solution(denoted by $\mathcal{P}^* = NULL$), then the algorithm reports no

```
input
   P : a multipath program;
   η : invariant template with parameters P
   h : a threshold
begin
   repeat
      valueArray := randomSampling(P, varList, n)
      Ψ := generatingLinearSystem(valueArray, η)
      P* := solvingLinearSystem(Ψ)
      if P* = 0 or P* = NULL
         return "no invariant for given template"
      n++;
   until P* is stable
   if isValidInvariant(P, η[P*/P], h) = true
    return "invariant:" η[P*/P]
end
```

Fig. 7. Algorithm for automated generation of loop invariants

invariants for the given template, otherwise the validity of the likely equality loop invariant $\eta[\mathcal{P}^*/\mathcal{P}]$ will be verified by calling $isValidInvariant(P, \eta[\mathcal{P}^*/\mathcal{P}], h)$, if $\eta[\mathcal{P}^*/\mathcal{P}]$ is a real loop invariant, then it is returned as output.

The algorithm for verifying the validity of the likely loop invariant $E(\overline{X}) = 0$

```
isValidInvariant(P, E(X̄) = 0, h)
input
    P : a multipath program;
    E(X̄) = 0 : a likely loop invariant
    h : a threshold
begin
    if E(X̄) has an l(l ≤ h)-height DFDT
      return true
    else
      return false
end
```

Fig. 8. Algorithm for automated verification of loop invariants

is described in Figure.8. According to theorem 1, the validity of loop invariant is verified by checking if $E(\overline{X})$ has an $l(l \leq h)$-height DFDT, when $E(\overline{X})$ has an $l(l \leq h)$-height DFDT, the algorithm decides $E(\overline{X}) = 0$ is a real loop invariant, when $E(\overline{X}) = 0$ has no $l(l \leq h)$-height DFDT, the algorithm decides $E(\overline{X}) = 0$ is not a real loop invariant.

5 Experimental Results

We evaluate our approach by generating equality loop invariants for those imperative programs tested in [8]. For each program, we have written a Mathematica program like cubicroot.m in Figure.5. Provided correct invariant templates and sufficiently random testings, the likely loop invariants generated by the written Mathematica programs are all real loop invariants. The validity of all the likely loop invariants are verified manually. The experimental results are listed in Table 2, the time taken for generating likely loop invariants are listed in the column labeled by time. The experiment is implemented on a laptop having Intel(R) Core(TM) M620 2.67GHz CPU and 3G memory.

Except the square root algorithm (Dijkstra,1976), all the tested algorithms in [8] that could be found by us are tested in the experiment. We now briefly describe the benchmark programs from Table 2. In most cases, we sample the values of the program variables after each loop iteration, for the Hardware Integer Division algorithm, we sample the values of program variables at the exit of the loop, because the program variables q and r in the given template are assigned values at the exit of the loop. In most cases, we use $RandomInteger[\{1, 1000\}]$ to generate random inputs for tested programs, for the Fermat's factorization algorithm and the Factoring Large Numbers algorithm, we use $RandomPrime[\{3, 1000\}]$

to generate random inputs, because these two programs require the odd integers as inputs.

The likely loop invariants are verified real loop invariants by checking if there exists finite DFDTs. For most likely loop invariants, their finite differences over all transitions are zero, in these cases, it is easily to verify the likely loop invariants are real loop invariants.

For the Wensleys algorithm program, whose multipath program abstraction is described in Fig.9., the DFDTs of the likely loop invariants are described in Fig.10. By theorem 1 , the likely loop invariants are real loop invariants.

$(a, b, d, y) =: (0, \frac{Q}{2}, 1, 0)$
while *true* **do**
  $\tau_1 : (b, d) := (\frac{b}{2}, \frac{d}{2})$
or
  $\tau_2 : (a, y, b, d) := (a + b, y + \frac{d}{2}, \frac{b}{2}, \frac{d}{2})$
od

Fig. 9. Abstraction of the Wensley's algorithm program

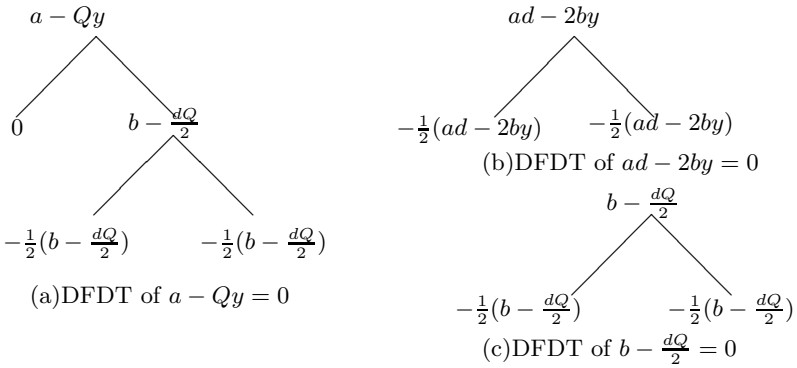


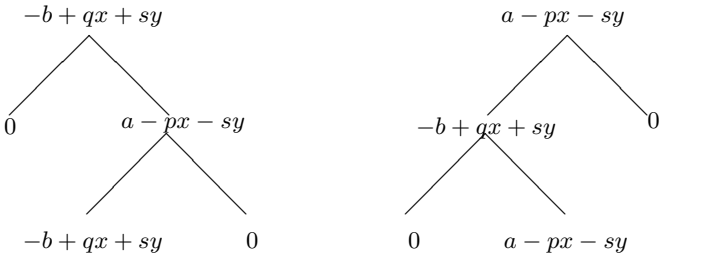
Fig. 10. some DFDTs of the Wensley's algorithm program

For the Extended GCD algorithm program, whose multipath program abstraction is described in Fig.11., some DFDTs of the likely loop invariants are described in Fig.12. By theorem 1 , both the likely loop invariants are real loop invariants.

The results shown in Table 2 demonstrate that our technique can be utilized to generate equality loop invariants efficiently.

$(a, b, p, q, r, s) =: (x, y, 1, 0, 0, 1)$
while *true* **do**
  $\tau_1 : (a, p, r) := (a - b, p - q, r - s)$
or
  $\tau_2 : (b, q, s) := (b - a, q - p, s - r)$
od

Fig. 11. Abstraction of the Extended GCD algorithm program



(a)DFDT of $-b + qx + sy = 0$ (b)DFDT of $a - px - sy = 0$

Fig. 12. some DFDTs of the Extended GCD algorithm program

6 Related Work

Many different approaches have been developed for inferring loop invariants. These approaches include:
(i)techniques based on abstract interpretation such as [1] [2].
(ii)template-based techniques, such as [3] [5] [6].
(iii)algebraic techniques such as [7] [8] [10] [11] [12].
(iv)dynamic methods such as [13] [14] [15] [16].
We discuss each of these four classes in more detail below.

Abstract Interpretation Techniques. Abstract interpretation is, roughly, a symbolic execution of programs over abstract domains that over-approximates the semantics of loop iteration. These techniques usually require a fixed point calculation. And some sophisticated widening and refinement techniques are employed to capture a precise over-approximation of fixed points. These techniques efficiently discover linear invariants, such as $\pm x \leq c$(Intervals), $x - y \leq c$(DBMs), $ax + by \leq c$ (where $a, b \in \{-1, 0, 1\}$, and c is an integer)[2], and $a_1x_1 + \cdots a_nx_n \leq c$ (where $a_1, \cdots, a_n$ and c are integers)[1].

Template-Based Techniques. Given an input template (i.e., parameterized form of invariant) provided by the user, template-based techniques find values for the parameters such that these instantiated templates correspond to inductive invariants. A method for generating linear invariants is proposed in [3].

Table 2. Experimental Results

Example	Templates	Result	time(s)
Division (Dijkstra,1976)	$c_1 + c_2 rem + c_3 quo.y + c_4 x = 0$	$c_4(-rem - quo.y + x) = 0$	0.003
Integer Square Root (Kirchner,1999)	$c_1 a + c_2 r^2 + c_3 r + c_4 x + c_5 = 0$	$c_4(-\frac{1}{2}a + \frac{1}{2}r^2 - \frac{1}{2}r + x) = 0$	0.005
Integer Square Root (Knuth,1998)	$c_1 + c_2 k + c_3 j + c_4 j^2 + c_5 m = 0$	$c_3(-1 - 2k + j) + c_5(-\frac{1}{4}j^2 + m - k - \frac{3}{4}) = 0$	0.01
Integer Cubic Root (Knuth,1998)	$c_1 + c_2 a + c_3 r + c_4 r^2 + c_5 r^3 + c_6 x + c_7 s = 0$	$c_6(-\frac{1}{4} - a + \frac{3}{4}r - \frac{3}{2}r^2 + r^3 + x) + c_7(-\frac{1}{4} - 3r^2 + s) = 0$	0.006
Sum of Powers n^5 (Petter,2004)	$c_1 x + c_2 y + c_3 y^2 + c_4 y^3 + c_5 y^4 + c_6 y^5 + c_7 y^6 = 0$	$c_7(-6x - \frac{1}{2}y^2 + \frac{5}{2}y^4 + 3y^5 + y^6) = 0$	0.02
Wensly's algorithm (Wegbreit,1974)	$c_1 b + c_2 dQ + c_3 a + c_4 Qy + c_5 ad + c_6 by = 0$	$c_2(-2b + dQ) + c_4(-\frac{1}{2}a + Qy) + c_6(-ad + by) = 0$	0.09
LCM-GCD computation (Dijkstra,1976)	$c_1 ab + c_2 ux + c_3 vy + c_4 = 0$	$c_3(-2ab + ux + vy) = 0$	0.006
Extended GCD (Knuth,1998)	$c_1 ps + c_2 qr + c_3 + c_4 b + c_5 qx + c_6 sy + c_7 a + c_8 px + c_9 ry + c_{10} x + c_{11} as + c_{12} br + c_{13} y + c_{14} bp + c_{15} aq = 0$	$c_3(-ps + qr + 1) + c_6(-b + qx + sy) + c_9(-a + px + ry) + c_{12}(x - as + br) + c_{15}(y - bp + aq) = 0$	0.014
Fermat's factorization (Knuth,1998)	$c_1 N + c_2 r + c_3 u + c_4 u^2 + c_5 v + c_6 v^2 = 0$	$c_6(4N + 4r + 2u - u^2 - 2v + v^2) = 0$	0.005
Square Root (Zuse,1993)	$c_1 q^2 + c_2 pr + c_3 a = 0$	$c_3(-q^2 + 2pr + a) = 0$	0.003
Binary Product (Knuth,1998)	$c_1 ab + c_2 xy + c_3 z = 0$	$c_3(-ab + xy + z) = 0$	0.006
Binary Product (Rodriguez-Carbonell and Kapur,2007b)	$c_1 abp + c_2 q + c_3 xy = 0$	$c_3(-abp - q + xy) = 0$	0.006
Binary Division (Kaldewaij,1990)	$c_1 A + c_2 bq + c_3 r = 0$	$c_3(-A + bq + r) = 0$	0.005
Hardware Integer Division (Manna,1974)	$c_1 y_2 + c_2 x_2 y_3 + c_3 x_1 y_3 + c_4 y_1 y_3 + c_5 x_2 y_3 y_4 = 0$	$c_2(-y_2 + x_2 y_3) + c_5(-x_1 y_3 + y_1 y_3 + x_2 y_3 y_4) = 0$	0.001
Hardware Integer Division (Sankaranaryanan,2004)	$c_1 y_2 + c_2 x_2 y_3 + c_3 y_1 y_3 + c_4 y_2 y_4 + c_5 y_3 x_1 + c_6 x_1 + c_7 r + c_8 x_2 q = 0$	$c_2(-y_2 + x_2 y_3) + c_5(-y_1 y_3 - y_2 y_4 + y_3 x_1) + c_8(-x_1 + r + x_2 q) = 0$	0.009
Factoring Large Numbers (Knuth,1998)	$c_1 d^2 q + c_2 rd + c_3 rpd + c_4 dq + c_5 r + c_6 n = 0$	$c_6(\frac{1}{4}d^2 q - \frac{1}{4}rd + rpd + \frac{1}{2}dq - \frac{1}{2}r + n) = 0$	0.009

Given an invariant template with undetermined coefficients, they obtain a set of non-linear constraints by Farkas lemma. The solutions to the constraints system yield the invariants. In [4], linear invariant templates (possibly disjunctive) and a set of second-order constraints are harnessed. The second-order constraints are converted to conjunctive normal form and reduced to first-order constraints by Farkas lemma. A satisfiability solver is employed for the first-order non-linear constraints to yield invariants. In [5], polynomial equality templates are used. By imposing that the template is invariant, they obtain a system of non-linear constraints by means of the Gröbner basis computation algorithm, and solving the constraints to yield invariants. In [6], the polynomial invariant generation problem is reduced to solving semi-algebraic systems and invariants are obtained by solving the semi-algebraic systems with the computer algebra tools DISCOVER and QEPCAD.

Algebraic Techniques. For programs with affine assignments, an precise interprocedural method for computing bounded degree polynomial equalities invariants has been proposed in [10]. They also propose a technique for discovering all the bounded degree polynomial invariants of programs with polynomial assignments and disequality guards in [11].

In [7], it is first shown that the set of polynomial loop invariants has the algebraic structure of an ideal, based on this connection, a fixed point procedure using operations on ideals and Gröbner basis constructions is proposed to find all polynomial invariants of simple loops. And it is proved that the fixed point procedure terminates in at most $m + 1$ iterations, where m is the number of program variables.

P-solvable loops are a family of imperative loops, for which test conditions in the loop and conditional branches are ignored, and the value of each program variable is expressed as a polynomial of the initial values of variables, loop counter, and some new variables where there are algebraic dependencies among the new variables. For P-solvable loops, a method for generating polynomial equality invariants is proposed and proven to be complete for some special cases[8]. In the approach, recurrence relations expressing the value of each program variable at the end of any iteration are formulated and solved exactly to yield the closed form for each loop variable. Loop counters are eliminated by Gröbner basis methods to yield invariants.

Dynamic methods. More recently, dynamic techniques have been applied to invariant inference. The Daikon approach of Ernst et al. [13] showed that dynamic inference is practical and sprung much derivative work (e.g., [14] [15] [16] and many others). Just like testing is quite effective and useful in practice, dynamic invariant inference is efficacious and many of the likely invariants are indeed sound.

7 Conclusions

We have presented a practical approach that generates equality loop invariants using random testing, constraint solving and verification. The main advantage

of the approach is that it produces linear equation constraints system, and the approach has no resort to Gröbner basis computing, recurrence relation solving, or fixed point computation, it is thus applicable to a broad class of imperative programs.

The main drawback of the method is that it requires the user to specify the shape of the desired loop invariants.

References

1. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) POPL 1978, pp. 84–96. ACM Press, Tucson (1978)
2. Mine, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
3. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear Invariant Generation Using Non-linear Constraint Solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
4. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI 2008, pp. 281–292. ACM Press, Tucson (2008)
5. Sankaranaryanan, S., Sipma, H.B., Manna, Z.: Non-Linear Loop Invariant Generation using Gröbner Bases. In: Jones, N.D., Leroy, X. (eds.) POPL 2004, pp. 318–329. ACM Press, Venice (2004)
6. Chen, Y., Xia, B., Yang, L., Zhan, N.: Generating Polynomial Invariants with DISCOVERER and QEPCAD. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 67–82. Springer, Heidelberg (2007)
7. Rodriguez-Carbonell, E., Kapur, D.: Generating All Polynomial Invariants in Simple Loops. J. of Symbolic Computation 42(4), 443–476 (2007)
8. Kovács, L.: Automated Invariant Generation by Algebraic Techniques for Imperative Program Verification in Theorema. PhD thesis, RISC, Johannes Kepler University Linz (2007)
9. Kovács, L.: Aligator: A Mathematica Package for Invariant Generation (System Description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 275–282. Springer, Heidelberg (2008)
10. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: Jones, N.D., Leroy, X. (eds.) POPL 2004, pp. 330–341. ACM Press, Venice (2004)
11. Müller-Olm, M., Seidl, H.: Computing polynomial program invariants. Inf. Process. Lett. 91(5), 233–244 (2004)
12. Müller-Olm, M., Petter, M., Seidl, H.: Interprocedurally Analyzing Polynomial Identities. In: Durand, B., Thomas, W. (eds.) STACS 2006. LNCS, vol. 3884, pp. 50–67. Springer, Heidelberg (2006)
13. Michael, D.E., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions of Software Engineering 27(2), 99–123 (2001)

14. Csallner, C., Tillman, N., Smaragdakis, Y.: DySy: dynamic symbolic execution for invariant inference. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE 2008, pp. 281–290. ACM Press, Leipzig (2008)
15. Perkings, J.H., Ernst, M.D.: Efficient incremental algorithms for dynamic detection of likely invariants. In: Taylor, R.N., Dwyer, M.B. (eds.) SIGSOFT FSE 2004, pp. 23–32. ACM Press, Newport Beach (2004)
16. Polikarpova, N., Ciupa, I., Meyer, B.: A comparative study of programmer-written and automatically inferred contracts. In: Rothermel, G., Dillon, L.K. (eds.) ISSTA 2009, pp. 93–104. ACM Press, Chicago (2009)

ConSMutate: SQL Mutants for Guiding Concolic Testing of Database Applications

Tanmoy Sarkar, Samik Basu, and Johnny S. Wong

Department of Computer Science, Iowa State University
{tanmoy,sbasu,wong}@iastate.edu

Abstract. Database applications are built using two different programming language constructs: one that controls the behavior of the application, also referred to as the *host language*; and the other that allows the application to access/retrieve information from the backend database, also referred to as the *query language*. The interplay between these two languages makes testing of database applications a challenging process. Independent approaches have been developed to evaluate test case quality for host languages and query languages. Typically, the quality of test cases for the host language (e.g., Java) is evaluated on the basis of the number of lines, statements and blocks covered by the test cases. High quality test cases for host languages can be automatically generated using recently developed concolic testing techniques, which rely on manipulating and guiding the search of test cases based on carefully comparing the concrete and symbolic execution of the program written in the host language. Query language test case quality (e.g., SQL), on the other hand, is evaluated using mutation analysis, which is considered to be a stronger criterion for assessing quality. In this case, several mutants or variants of the original SQL query are generated and the quality is measured using a metric called mutation score. Higher mutation score indicates higher quality for the test cases. In this paper we present a framework, called *ConSMutate*, which guides concolic testing using mutation analysis for test case generation for database applications. The novelty of the framework is that it ensures that the test cases are of high quality not only in terms of coverage of code written in the host language, but also in terms of mutant detection of the queries written in the query language. We present a prototype implementation of our technique and show its advantages using two non-trivial case studies.

Keywords: Database Applications, Automatic Test Case Generation, Concolic Testing, Mutation Analysis.

1 Introduction

Background. Automated test case generation techniques have been proposed and developed to minimize human effort in testing. Existing approaches [9,17,18] for test case generation often use block or branch coverage of the application as a primary criterion for ensuring the quality of the generated test cases. Test cases

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 462–477, 2012.

achieving high block or branch coverage certainly increases the confidence on the quality of the application under test; however, coverage cannot be argued as a sole criterion for effective testing. Specifically, test cases generated solely on the basis of coverage criteria may not catch simple errors in application logic; e.g., errors may occur due to incorrect usage of relational operators (e.g., $<$ instead of $\leq$, $=$ instead of $==$).

In such scenarios, mutation testing [15] has been proven effective for assessing the quality of the generated test inputs. In mutation testing, the original program is modified slightly based on typical programming errors (such as those mentioned above). The modified version is referred to as the *mutant*. Each mutant is evaluated against the set of test cases to determine if it can be distinguished from the original program by the test results. If so, the mutant is said to be *killed*. The quality of the test cases is measured by a metric called *mutation score*-defined as the ratio between the number of mutants killed and the total number of non-equivalent mutants (mutants which are not semantically identical to the actual program). High mutation score implies that that test cases can be used to identify and gain valuable insights to the existence of typical programming errors.

Driving Problem. Typically, test case generation relies on ensuring a high degree of (code, block or branch) coverage, and mutation testing is performed separately. If the mutation score of the generated test cases is low, new test cases are generated and mutation analysis is performed again. This results in unnecessary delay and overhead in identifying the high quality test cases, where quality is attributed to both coverage and mutation scores.

In this paper, we propose and develop a test case generation technique which addresses the above problem.

Our Solution. We present a framework, ConSMutate, capable of automatically generating high quality test cases for database applications. It relies on *Con*crete and *S*ymbolic execution of the application program written in host language (language in which the database application is coded) and uses *Muta*tion analysis of database-queries written in embedded language to guide the generation of high quality of test cases. At its core, our framework uses Pex [19], a state-of-the-art dynamic symbolic execution engine (DSE),[1] to generate test inputs for an application program with high code coverage. For every new test case, we measure the mutation score of the test case. If the mutation score of the test case is below the pre-specified threshold, our framework analyzes the path constraints (necessary for coverage) and mutation-killing constraints (necessary for high mutation score), and uses a constraint solver to automatically identify a new test case with high quality.

Contributions: The contributions of our work are summarized as follows:

1. To the best of our knowledge, this is the first approach that combines coverage analysis and mutation analysis in automatic test case generation for

[1] Our framework is not tightly coupled to the Pex engine; any engine that is based on concolic testing can be deployed in our framework.

database applications which involve two different languages: host language and embedded query language.
2. The impact of our proposed framework is that it reduces the overhead of high quality test case generation by avoiding test cases with low coverage and low mutation scores.
3. We evaluate the practical feasibility and effectiveness of our proposed framework by applying it on two real database applications. We compare our method against Pex [19], a white-box testing tool for .NET from Microsoft Research, and show that our method generates test cases with higher code coverage and higher mutation score compared to the ones generated by Pex.

Organization. The rest of the paper is organized as follows. Section 2 presents a simple example that is used to motivate our work and is used to discuss the salient aspects of our technique. Section 3 discusses existing work related to our work. Section 4 constitutes the main body of our paper and discusses in detail the proposed technique. Section 5 presents empirical evaluation of our technique followed by future avenues of research in Section 6.

2 Motivating Example

Consider the pseudocode in the above procedure chooseCoffee. It represents a typical database application; it takes as two input parameters x and y, creating different query string depending on the valuation of the parameters which guides the control path in the application. Assume that one of the database tables `coffees` contains three entries as shown in Table 1.

Pex generates three test cases, e.g., (0, 0), (11, 0) and (11, 2), taking into consideration the branch conditions in the application program. The first and the second values in the tuple represent the valuations of x and of y respectively. These test cases cover all branches present in the program. However, as the

Algorithm 1. Sample Pseudocode for Database Application

```
1: procedure chooseCoffee(x, y)
2:     String q = " ";
3:     if x>10 then
4:         y++;
5:         if y≤ 2 then
6:             q = "SELECT cof_name FROM coffees WHERE price =" + y + ";";
7:         else
8:             q = "SELECT cof_name FROM coffees WHERE price ≤" + y + ";";
9:         end if
10:    end if
11:    if q != " " then
12:        executeQuery(q);
13:    end if
14:    return;
15: end procedure
```

Table 1. Table **coffees**

cof_name	sup_id	price
Colombian	101	1
French_Roast	49	2
Espresso	150	10

database is not taken into consideration for the test case generation, the test cases are unlikely to kill all mutants corresponding to the query being executed. For instance, the test case (11, 0) results in the execution of the query generated at Line 6.

The executed query

$$\text{SELECT cof\_name FROM coffees WHERE price} = 1$$

generates the result `Colombian` using the *coffees* table. A mutant of this query

$$\text{SELECT cof\_name FROM coffees WHERE price} \leq 1$$

is generated by slightly modifying the "WHERE" condition in the query (mimicking typical programming errors). The result of the mutant is also `Colombian`. That is, if the programmer makes the error of using the equal-to-operator in the "WHERE" condition instead of the intended less-than-equal-to operator, then that error will go un-noticed if test case $(11, 0)$ is used. Note that there exists a test case $(11, 1)$ which can distinguish both the mutants from the original query without compromising branch coverage. We will show in section 4 that our framework successfully identifies such test cases automatically.

3 Related Work

In recent years, automatic generation of test cases for database application [9,14,18] has attracted much attention from researchers. There are two main approaches: generating database states from scratch [9,18] and using existing database states [14]. The objective of both these approaches is to achieve *high branch coverage*.

The above techniques do not consider mutation analysis (also known as mutation testing) as a way to measure the quality of the test cases. However, mutation testing has been argued to be an effective indicator of quality of test cases [3]. Also, [14] has shown that mutation testing can be superior to common code coverage in evaluating the quality of test cases.

Mutation testing [8] is a fault-based testing approach. In mutation testing a large number of alternative programs called mutants is generated by transforming the original code using a set of pre-defined rules (mutation operators) that are developed to introduce simple syntactic changes based on errors that programmers typically make or to force common testing goals. Each mutant is executed with the test data, and when it produces a different end result (strong mutation testing [7]) or a different intermediate state (weak mutation testing [10]), the mutant is said to be killed. A test case is said to be effective if it kills some mutants

that have not yet been killed by any of the previously executed test cases. Some mutants *always* produce the same output as the original program, so no test case can kill them. These mutants are said to be *equivalent mutants*. After executing a test set over a number of mutants, the mutation score is defined as the ratio between dead mutants and the number of non-equivalent mutants. Mutation testing was primarily developed for programming languages like Fortran and Ada [22]. For database application, SQL mutation operators have been developed [20] and coverage criteria of isolated SQL statements [21] have been defined separately. Mutation Analysis is not only used in code-based testing approaches (white-box testing), but it is also extensively used in formal model-based testing approaches [1,13]. These approaches are proven to be effective where formal verification is not feasible and are able to distinguish whether an implementation refines a faulty specification. Jia and Harmen [11] present a detailed analysis and valuable insights of current development trends in mutation testing.

In contrast to the above techniques, our proposed method aims to combine the coverage criteria and mutation analysis in such a way that test cases with high coverage and high mutation score are generated automatically. The primary challenge addressed in our work is the consideration of database applications where the coverage criteria depends on the application language while the mutation score relies only on the embedded query language.

In this work, we have assumed that a sample/test database table is provided a priori; therefore, the highest achievable mutation score depends on the adequacy/completeness of the database table. For instance, an empty database table is unlikely to help in generating test cases with high mutation score (results to all queries may be empty). It should be noted that a large number of works [4,5,6,12] are concentrated on identifying the "sufficient" database table for generating test cases with high mutation score for the database queries. Our work is orthogonal to these works; however, our method can directly benefit from their results.

4 ConSMutate Test Case Generator for DB-Applications

Figure 1 presents the ConSMutate framework. It has two main modules, *Application Branch Analyzer* and *Mutation Analyzer*. The Application Branch Analyzer takes the program under test and the sample database as inputs, and generates test cases and the corresponding path constraints. It uses Pex [19], a dynamic symbolic execution engine (other engines like concolic testing tools [17] can also be used), to generate test cases by carefully comparing the concrete and symbolic execution of the program. After exploring each path, the Mutation Analyzer module performs mutation quality analysis using mutation score. If the mutation score is low, Mutation Analyzer generates a new test case for the same path whose *quality* is likely to be high. The steps followed in our framework for generating test cases are presented in the following subsections.

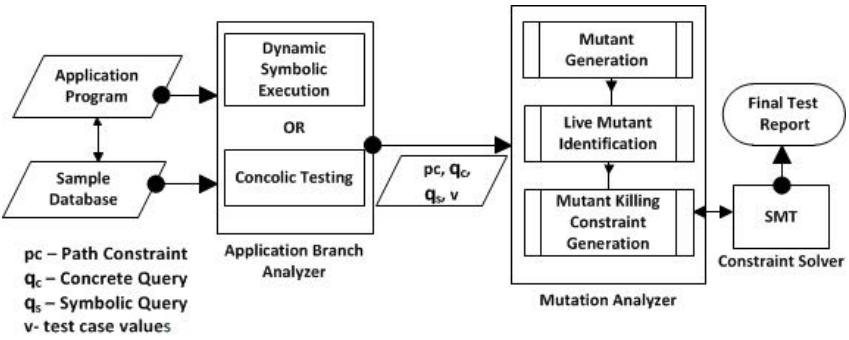


Fig. 1. Framework for ConSMutate

4.1 Generation of Test Cases and Associated Path Constraints Using Application Branch Analyzer

In the first step, the framework uses the *Application Branch Analyzer* module to generate a test case value v and the associated path constraints. It results in a specific execution path constraint (say, pc) of the application, which in turn results in a database query execution (if the path includes some query). The executed query is referred to as the concrete query q_c and the same without the concrete values (with the symbolic state of the input variable) is referred to as the symbolic query q_s. The path constraints refer to the conditions which must be satisfied for exploring the execution path in the application.

Going back to the example in Section 2, Application Branch Analyzer (Pex in our case) generates a test case $v = (11, 0)$, i.e., $x = 11$ and $y = 0$. This results in an execution path with path constraints $pc = (x > 10) \wedge (y + 1 \leq 2)$. It also results in a symbolic query and a corresponding concrete query:

Symbolic q_s: SELECT cof_name FROM coffees WHERE price = y_s
Concrete q_c: SELECT cof_name FROM coffees WHERE price = 1

where y_s is related to program input y as $y_s = y + 1$ at line 6 (see the example program in Section 2).

4.2 Deployment of Mutation Analyzer

After exploring a path of the program under test, ConSMutate forwards pc, q_c, q_s and v to *Mutation Analyzer* to evaluate the quality of the generated test case in terms of mutation score.

4.2.1 Generation of Mutant Queries

In Mutation Analyzer, the obtained concrete query q_c is mutated to generate several mutants q_m(s). The mutations are done using pre-specified mutation functions in the *Mutant Generation* module.

It is generally agreed upon that a large set of mutation operators may generate too many mutants which, in turn, exhaust time or space resources without

Table 2. Sample mutant generation rules and mutant killing-constraints

Mutation Rule	Original	Mutant	Mutant Killing-constraint
Relational Operator Replacement (ROR),$\alpha, \beta \in$ ROR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$((C_1\ \alpha\ C_2) \wedge \neg(C_1\ \beta\ C_2))$ $\|\|$ $(\neg(C_1\ \alpha\ C_2) \wedge (C_1\ \beta\ C_2)))$
Logical Operator Replacement (LOR),$\alpha, \beta \in$ LOR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$(C_1\ \alpha\ C_2) \neq (C_1\ \beta\ C_2)$
Arithmetic Operator Replacement (AOR),$\alpha, \beta \in$ AOR and $\alpha \neq \beta$	$C_1\ \alpha\ C_2$	$C_1\ \beta\ C_2$	$(C_1\ \alpha\ C_2) \neq (C_1\ \beta\ C_2)$

offering substantial benefits. Offutt et al. [16] proposed a subset of mutation operators which are approximately as effective as all 22 mutation operators of Mothra, a mutation testing tool [8]. They are referred as *sufficient mutation operators*. In our context, we are specifically focused on SQL mutants. We have identified *six* mutation operators by comparing SQL mutation operators developed in [20] with the sufficient set of mutation operators mentioned in [16]. We refer to these six rules as the *sufficient set of SQL mutation operators*, sufficient to identify logical errors present in the `WHERE` and `HAVING` clauses.

ConSMutate uses these mutation operators in generating mutants. It should be noted here that new mutation operators can be considered and incorporated in mutation generation module in ConSMutate as and when needed. Table 2 (first three columns) presents three such mutation generation rules for relational, logical and arithmetic operators. Going back to the example in section 2 one of the mutants of the symbolic q_s is

$$q_m\text{: SELECT cof\_name FROM coffees WHERE price} \leq y_s$$

In the above transformation, α is "=" (equality relational operator) and β is "$\leq$" (less-than-equal-to relational operator) as per the rule in the first row, second and third columns of Table 2.

4.2.2 Identification of Live Mutants

Using the test case under consideration, the live mutants are identified. Live mutants are the ones whose results do not differ from that of the concrete query in the context of the given database table. The above mutant q_m is live under the test case $v = (11, 0)$ as it results in a concrete query

$$\text{SELECT cof\_name FROM coffees WHERE price} \leq 1$$

Recall that $y_s = y + 1$ and $y = 0$ for the test case $(11, 0)$ when the query is constructed in Line 6 (see program in Section 2). The above query and the concrete query q_c produce the same result for the given database table (Table 1). Therefore, q_m is live under the test case $(11, 0)$.

4.2.3 Generation of Mutant Killing Constraints

A new set of constraints θ is generated in *Mutant Killing Constraint Generation* module in two steps:

1. Generation of constraint from queries
   - the symbolic query q_s and its concrete version q_c
   - the live mutants (q_m's) computed in the previous step
   - the concrete and symbolic state of the program inputs which is affected by the test cases
2. Incorporation of path constraints (pc) to ensure the same path is explored and therefore the same set of queries are executed

Generation of Constraint from Queries. We proceed by capturing the concrete and symbolic queries executed in the path explored by the given test case. This is done using Pex API methods `PexSymbolicValue.ToString(..)` and `GetRelevantInputNames(..)`. We decompose concrete and symbolic query using a simplified SQL parser and get their `WHERE` conditions, which we assume to be in conjunctive normal form.

Identification of Query Conditions. We then identify the conditions that resulted in a mutant query and their relationship with the test inputs (or program inputs). We refer to the conditions obtained from the original query as the *original query-condition* and, likewise, the conditions obtained from the mutant query as the *mutant query-condition*.

For the *concrete versions* of the original and the mutant query-condition, we identify the satisfiable valuations of the database attribute. For instance, in our running example, the original query-condition is $price = 1$ and the mutant query-condition is $price \leq 1$. We query the database to find one valuation of *price* which satisfies these conditions. Note that the same valuation of *price* will satisfy both the conditions as we are considering the live mutants. In our running example, the original query-condition and the mutant query-condition are satisfied when the value of *price* is set to 1 (see Table 1).

Using the above and the *symbolic versions* of the original and the mutant query-conditions, we identify the relationship between the valuations of database attributes and the test inputs. For instance, in our running example, the original symbolic query-condition is $price = y_s$ and the mutant symbolic query-condition is $price \leq y_s$. We also know that y_s is set to $y + 1$ (y is one of the test inputs) and *price* is set to 1. Therefore, the relationship between the valuations of the database attribute *price* and the test input y is $1 = y + 1$ in the original query-condition and $1 \leq y + 1$ in the mutant query-condition. We will use these relationships/conditions for generating the mutant killing constraint; we refer to them as the *original input-condition* and the *mutant-input condition*.

Identification of Mutation Points. The original and the mutant input-conditions are compared to identify the mutation point (the point at which the original input-condition and the mutant input-condition differ). Depending on the mutation point, a corresponding mutant killing constraint rule is triggered.

For complex conditions, ConSMutate uses a binary search algorithm to identify the mutation point. As an example, the original condition $(C_1 \leq C_2) \wedge (C_3 \leq C_4)$ can have a mutant $(C_1{=}C_2) \wedge (C_3 \leq C_4)$. ConSMutate first looks at the outmost level and finds that the logical operators remain the same for both

of these expressions. It recursively looks at the left and right sub-conditions of these expressions and identifies the mutation point. In this case the mutation point is at left-hand side i.e., $(C_1 \leq C_2)$ and $(C_1{=}C_2)$.

Identifyication of Mutant Killing Constraints for Conditions: Finally, for the original input-condition and its mutant, a mutant killing constraint is generated following the rules in Table 2. Satisfaction of the mutant killing constraint results in an assignment to the test inputs which satisfies (resp. does not satisfy) the original input-condition and does not satisfy (resp. satisfies) the mutant input-condition. For instance, for our running example, the mutant killing constraint is $[(1 = y + 1) \wedge (1 \not\leq y + 1)] \vee [(1 \neq y + 1) \wedge (1 \leq y + 1)]$ (using ROR rule from Table 2).

Incorporation of Path Constraints. We extract *path constraints* (*pc*) from Pex and conjunct them with the mutant killing constraint generated above to construct θ. This is necessary to ensure that any satisfiable assignment of test inputs results in exploration of the same execution path. In our running example, the path constraint is $(x > 10) \wedge (y + 1 \leq 2)$. The conjunction result will be θ as shown below.

$$\theta : (x > 10) \wedge (y + 1 \leq 2) \wedge \\ [((1 = y + 1) \wedge (1 \not\leq y + 1)) \vee ((1 \neq y + 1) \wedge (1 \leq y + 1))]$$

4.3 Deployment of Constraint Solver: Finding Satisfiable Assignment for θ

The constraint θ is checked for satisfiability to generate a new test case. We use the SMT solver named Yices[2] for this purpose. Other high performance constraint solvers like Z3[3] can be used in the constraint solver module (Figure 1). If θ is satisfied, then certain valuations of the inputs to the application are identified, which is the new test case v'. This new test case v' is guaranteed to explore the same execution path as explored due to test case v. Furthermore, some mutants that were left "live" by v are *likely to be* "killed" by v'. Therefore, it is necessary to check whether v' indeed kills the live mutants; if not, SMT solver is used again to generate a new satisfiable assignment for θ (including the negation of the previously generated value), which results in a new test case v''. This iteration is terminated after certain pre-specified times (e.g., 10) or after all live mutants are killed (whichever happens earlier). It should be noted that if the live mutant is equivalent to the original query in the context of the database table, then no new test case can differentiate between the mutant and the original query. Therefore, we use a pre-specified limit to the number of iterations after which we terminate the process.

Going back to our running example, when the SMT solver generates a satisfiable assignment $x = 11, y = 1$ for the mutant killing constraint θ (see above), the

[2] http://yices.csl.sri.com/

[3] http://research.microsoft.com/en-us/um/redmond/projects/z3/

Table 3. Mutants and results for test case (11, 1)

Query	Concrete Query	Result
q_c	SELECT cof_name FROM coffees WHERE price = 2	French_Roast
q_m	SELECT cof_name FROM coffees WHERE price ≤ 2	Colombian, French_Roast

new test case $v' = (11, 1)$ successfully kills the live mutant q_m by distinguishing its result from the original query result, as shown in Table 3. The above steps (starting from Section 4.1) are iterated to generate new test cases that explore different execution paths of the program. This iteration continues until all possible branches are covered following the method used by Pex.

4.4 Correctness Criteria of ConSMutate

Theorem 1. *For any path explored by a test case t_0 with path constraint pc, if the symbolic query executed along the path is q_s and if the live mutant is q_m, the set of satisfiable assignments for the mutant killing constraint θ as obtained by ConSMutate is a superset of the test cases that can kill the mutant.*

Proof. A test case can be viewed as a mapping of variables (inputs to programs) to values. We will denote this mapping as $t : [\bar{x} \mapsto \bar{v}]$, where t is a test case, $\bar{x}$ is an ordered set of inputs/variables and $\bar{v}$ is an ordered set of valuations[4].

We prove the above theorem by contradiction. We assume that there exists a test case t that can kill the mutant q_m; however it is not a satisfiable assignment for θ, denoted by $t \not\models \theta$.

As the test case t can kill the mutant q_m, it must satisfy the path constraint pc, which is necessary to explore the path where the original query q_s is generated and executed. Recall that θ contains a conjuct pc. Therefore, $t \not\models \theta_1$, where $\theta = pc \wedge \theta_1$.

Next, let us consider the construction of θ_1. WLOG, consider that there is one mutation point in the `WHERE` clause of q_s and q_m. Let the `WHERE` clause be $db_{var}\ \mathcal{R}\ x$, where db_{var} is a database variable, $\mathcal{R}$ is a relational operator and x is an input to the program (x can be a program variable dependent indirectly on the program input). Let the mutant q_m has the `WHERE` clause transformed by altering $\mathcal{R}$ to $\mathcal{R}'$. The original test case t_0 results in the valuation of x for which the `WHERE` clauses of q_s and q_m, i.e., $db_{var}\ \mathcal{R}\ x$ and $db_{var}\ \mathcal{R}'\ x$, produces the same set of results.

Therefore, $\theta_1 = \theta_{11} \vee \theta_{12}$, where

$$\theta_{11} = (v_0\ \mathcal{R}\ x) \wedge \neg(v_0\ \mathcal{R}'\ x)$$
$$\theta_{12} = \neg(v_0\ \mathcal{R}\ x) \wedge (v_0\ \mathcal{R}'\ x)$$

and $t_0 : [x \mapsto v_0]$.

As per our assumption, $t \not\models \theta_1$, i.e., $t \not\models \theta_{11}$ and $t \not\models \theta_{12}$. In other words, for $t : [x \mapsto v]$, both

[4] When the ordered set contains one variable, we denote test case t as $t : [x \mapsto v]$

$$(v_0 \ \mathcal{R} \ v) \wedge \neg(v_0 \ \mathcal{R}'v) \\ \neg(v_0 \ \mathcal{R} \ v) \wedge (v_0 \ \mathcal{R}'v) \tag{1}$$

evaluate to false.

Case-Based Argument: Consider that $\mathcal{R}$ is the equality relation $=$. Let the mutation rule result in $\mathcal{R}'$ equal to $\neq$ relation. It is immediate that at least one of the formulas in Equation 1 must be satisfiable (specifically the second formula must be satisfiable when $\mathcal{R}$ is $=$ relation). Therefore, our assumption that t is not a satisfiable assignment of θ is contradicted.

Next consider that the mutation rule resulted in $\mathcal{R}'$ to be $\leq$ relation. Note that $db_{var} = v_0$ and $db_{var} \leq v_0$ in the `WHERE` clause of the original and the mutant queries, respectively, produced equivalent/ indistinguishable results for the test case t_0; on the other hand, $db_{var} = v$ and $db_{var} \leq v$ in the `WHERE` clause of the original and the mutant queries, respectively, produced non-equivalent/indistinguishable results for the test case t. As $v \neq v_0$ (in which case the test cases will become identical), there are two possibilities: $v < v_0$ and $v_0 < v$.

If $v < v_0$, then the `WHERE` clause conditions $db_{var} \leq v$ would have produced results equivalent to the ones produced by $db_{var} = v$. This is because $db_{var} \leq v_0$ and $db_{var} = v_0$ produce equivalent results. However, as t can kill the mutant, the results produced by the valuation v for the original and the mutant clauses must be different. Therefore, $v < v_0$ does not hold. Proceeding further, $v_0 < v$ implies that the second formula in Equation 1 is satisfied, which leads to contradiction of our assumption that t does not satisfy θ.

Similar contradictions can be achieved, and the theorem statement can be proved for other operations. □

5 Preliminary Results

5.1 Evaluation Criteria

ConSMutate can utilize any DSE-based test generation tools (e.g., Pex [19] in .NET applications) to generate high quality test cases for database applications, where quality is attributed to both high coverage criteria and high mutation score. We evaluate the benefits of our approach from the following two perspectives:

1. What is the percentage increase in code coverage by the test cases generated by Pex compared to the test cases generated by ConSMutate in testing database applications?
2. What is the percentage increase in mutation score of test cases generated by Pex compared to the ones generated by ConSMutate in testing database applications?

We first run Pex to generate test cases (different valuations for program inputs) for methods with embedded SQL queries in two open source database applications. We record the mutation score and code coverage percentage achieved by

Table 4. Method names and corresponding Program Identifiers

UnixUsage		RiskIt	
Program Identifier(s)	**Method(s)**	**Program Identifier(s)**	**Method(s)**
1	courseIdExists	10	getOneZipcode
2	courseNameExists	11	filterMaritalStatus
3	getCourseIDByName	12	filterZipcode
4	getCourseNameByID	13	getValues
5	isDepartmentIdValid		
6	isRaceIdValid		
7	getDeptInfo		
8	deptIDExists		

Pex. Next we apply ConSMutate to generate test cases for the same methods and record the corresponding mutation score and code coverage statistics. The experiments are conducted on a PC with a 2GHz Intel Pentium CPU and 2GB memory running the Windows XP operating system.

5.2 Evaluation Test-Bed

Our empirical evaluations are performed on two open source database applications: *UnixUsage*[5] and *RiskIt*[6]. UnixUsage is a database application where queries are written against the database to display information about how users (students), who are registered in different courses, interact with the Unix systems using different commands. The database contains 8 tables, 31 attributes, and over a quarter million records. RiskIt is an insurance quote application which makes estimates based on users' personal information, such as zipcode. It has a database containing 13 tables, 57 attributes and over 1.2 million records[7]. Both applications are written in Java with backend Derby. To test them in the Pex environment, we convert the Java source code into C# code using a tool called Java2CSharpTranslator[8]. Since Derby is a database management system for Java and does not adequately support C#, we retrieve all the database records and populate them into Microsoft Access 2010. We also manually translate those original database drivers and connection settings into C# code.

Table 4 presents the methods in each of the test applications. The program identifiers 1–8 and 10–13 will be used to present our results in the rest of the sections.

5.3 Summary of Evaluation

Figure 2 shows the results of our evaluation. The graph compares the performances of Pex and ConSMutate in terms of achieving quality. The x-coordinates in the graph represent the Program Identifiers for different methods for UnixUsage and RiskIt as mentioned in table 4. The y-axis represents the Quality(%) in terms of Block Coverage and Mutation Score achieved by Pex and ConSMutate for various program identifiers.

[5] http://sourceforge.net/projects/se549unixusage
[6] https://riskitinsurance.svn.sourceforge.net
[7] http://webpages.uncc.edu/~kpan/coverageCriteria.html
[8] http://sourceforge.net/projects/j2cstranslator/

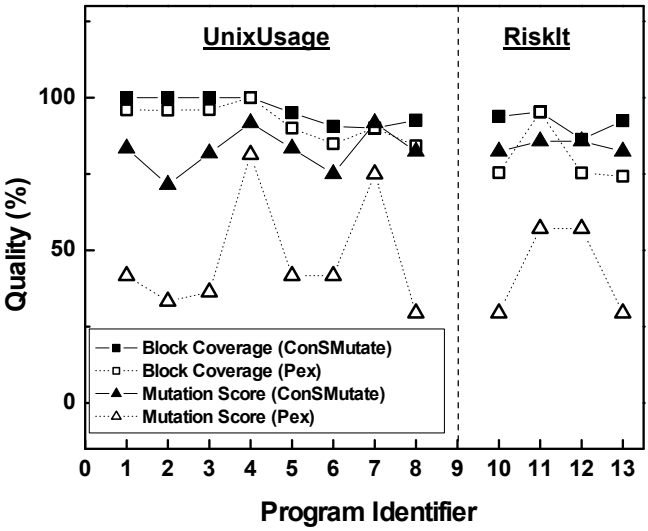


Fig. 2. Comparison between Pex and ConSMutate in terms of quality

5.3.1 Evaluation Criterion 1: Coverage Benefit

Figure 2 (points shown in square) demonstrates the block coverage achieved by Pex and ConSMutate for both of the applications. Although Pex has achieved good block coverage as expected, ConSMutate has successfully achieved more than 10% improvement in coverage in case of various methods (program identifiers in figure 2). The reason for this is that Pex cannot generate sufficient program inputs to achieve higher code coverage, especially when program inputs are directly or indirectly involved in embedded SQL statements. ConSMutate does not suffer from this drawback, as it considers database states and the results of generated queries and their execution results.

5.3.2 Evaluation Criterion 2: Mutation Score Benefit

Figure 2 (points shown in triangle) also demonstrates the mutation score achieved by Pex and ConSMutate for the test applications. The mutation score of test cases generated by ConSMutate is always higher than the mutation score of test cases generated by Pex. The increase in mutation score ranges from around 10% to 50%. We can see less increase in mutation score for methods like getCourseNameByID, getDeptInfo in Unix-Usage (program identifiers 4 and 7 in figure 2). Manual inspection reveals the fact that the improvement in mutation score is less for methods where the number of generated mutants are fewer than other methods.

The mutation scores achieved by ConSMutate are sometimes less than 100%, because the test cases generated by ConSMutate are *likely* to kill mutants and therefore may not be always successful. Figure 2 presents the mutation score achieved by ConSMutate by just performing constraint solving once (see Section 4.2). If the mutant is not killed by the test case obtained after one iteration of constraint solving, additional iterations of constraint solving can be done.

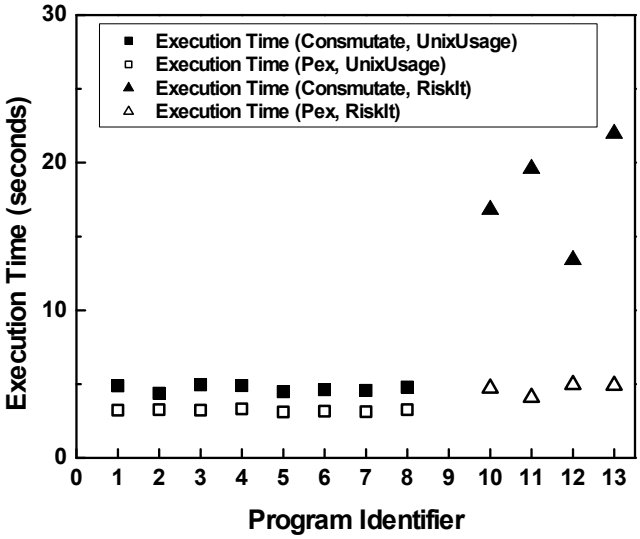


Fig. 3. Execution time comparison between Pex and ConSMutate

In our evaluation we do not eliminate equivalent mutants. We calculated mutation score as the number of mutants killed divided by total number of generated mutants. Note that there are a number of equivalent mutants for most of the cases and if we exclude these equivalent mutants, ConSMutate could achieve even higher mutant-killing ratios. Manual inspections show that the mutation scores achieved by ConSMutate are less than 100% because of the existence of equivalent mutants and because the database tables provided are not always sufficient to kill all the mutants.

5.4 Execution Time Overhead

As ConSMutate involves the database and utilizes a constraint solver for generating high quality test cases, there is obviously a penalty in terms of execution time. In this section, we show that the execution time overhead is not prohibitively large and therefore ConSMutate can be used effectively for test case generation for practical applications.

Figure 3 compares the execution times of Pex and ConSMutate. The x-coordinates in the graph represent the different Program Identifiers for UnixUsage and RiskIt as mentioned in Table 4. The y-axis represents time. For UnixUsage, the execution time of ConSMutate is approximately 1.3 times that of Pex. The increase in time is due to multiple mutant query execution and subsequent comparison of the large result sets returned by them from the backend database (more than 0.25 million records for UnixUsage). Multiple mutant executions are required in our framework in order to identify live mutants.

In the case of RiskIt, the increase in database size is *five*-times more than UnixUsage. As a result, the total execution time increases by five times (maximum

for method identified by program 13). Optimizing multiple query execution is an open research problem and several research works in this area [2,23] propose effective techniques which can reduce the total execution time by a considerable amount. Incorporating such techniques in our framework is not in the scope of our current objective but can be done easily to further improve the execution time.

6 Conclusion and Future Work

In this paper, we have proposed a framework called ConSMutate that combines coverage analysis and mutation analysis in automatic test case generation for database applications using a given database state. Our initial experiments show the effectiveness and practical applicability of the approach. Moreover, our framework is generic, and therefore new coverage-based and mutation generations techniques can be easily incorporated and evaluated in the framework.

Killing SQL mutants depends partially on choosing the right test cases and partially on the current database state. Since the framework relies on identifying important control-path constraints of the application and the constraints for killing mutants, the constraint generated so far may result in a satisfiable assignment that will not be able to kill all the mutants. However, during test case generation, ConSMutate captures the relationship between the database variables present in the queries and the program inputs. This can help to generate a new constraint, *table constraint*, which is the acceptable set (or range) of values for the program inputs with respect to the given database state.

As an extension of our current work, we plan to investigate the role of table constraints in conjunction with path constraints and mutant killing constraints to identify new test cases. This new expression will allow the constraint solver to identify test cases whose likelihood of killing mutant is higher. Also, we plan to investigate the insights provided by these constraints and develop a technique which will help in obtaining the necessary and sufficient database states for generating test cases with pre-specified (even 100%) coverage and mutation scores.

References

1. Aichernig, B.K., Brandl, H., Jöbstl, E., Krenn, W.: Efficient mutation killers in action. In: ICST, pp. 120–129 (2011)
2. Andrade, H., Kurc, T., Sussman, A., Saltz, J.: Optimizing the execution of multiple data analysis queries on parallel and distributed environments. IEEE Trans. Parallel Distrib. Syst. 15(6), 520–532 (2004)
3. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: ICSE, pp. 402–411 (2005)
4. Binnig, C., Kossmann, D., Lo, E.: Reverse query processing. In: Chirkova, R., Dogac, A., Özsu, M.T., Sellis, T.K. (eds.) ICDE, pp. 506–515. IEEE (2007)
5. Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J.: An AGENDA for testing relational database applications. Softw. Test., Verif. Reliab. 14(1), 17–44 (2004)

6. Chays, D., Shahid, J., Frankl, P.G.: Query-based test generation for database applications. In: Giakoumakis, L., Kossmann, D. (eds.) DBTest, p. 6. ACM (2008)
7. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer 11(4), 34–41 (1978)
8. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Software Eng. 17(9), 900–910 (1991)
9. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Rosenblum, D.S., Elbaum, S.G. (eds.) ISSTA, pp. 151–162. ACM (2007)
10. Howden, W.E.: Weak mutation testing and completeness of test sets. IEEE Trans. Software Eng. 8(4), 371–379 (1982)
11. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. 37(5), 649–678 (2011)
12. Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, S.: Query-aware test generation using a relational constraint solver. In: ASE, pp. 238–247. IEEE (2008)
13. Krenn, W., Aichernig, B.K.: Test case generation by contract mutation in spec#. Electr. Notes Theor. Comput. Sci. 253(2), 71–86 (2009)
14. Li, C., Csallner, C.: Dynamic symbolic database application testing. In: Babu, S., Paulley, G.N. (eds.) DBTest. ACM (2010)
15. Offutt, A.J., Jin, Z., Pan, J.: The dynamic domain reduction procedure for test data generation. Softw., Pract. Exper. 29(2), 167–193 (1999)
16. Offutt, A.J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: Basili, V.R., DeMillo, R.A., Katayama, T. (eds.) ICSE, pp. 100–107. IEEE Computer Society/ACM Press (1993)
17. Sen, K.: Concolic testing. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) ASE, pp. 571–572. ACM (2007)
18. Taneja, K., Zhang, Y., Xie, T.: MODA: Automated test generation for database applications via mock objects. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) ASE, pp. 289–292. ACM (2010)
19. Tillmann, N., de Halleux, J.: Pex–White Box Test Generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008)
20. Tuya, J., Cabal, M.J.S., de la Riva, C.: Mutating database queries. Information & Software Technology 49(4), 398–417 (2007)
21. Tuya, J., Cabal, M.J.S., de la Riva, C.: Full predicate coverage for testing sql database queries. Softw. Test., Verif. Reliab. 20(3), 237–288 (2010)
22. Voas, J.: Software fault injection: Growing "safer" systems. In: IEEE Aerospace Conf., vol. 2, pp. 551–561 (February 1997)
23. Weng, L., Çatalyürek, Ü.V., Kurç, T.M., Agrawal, G., Saltz, J.H.: Optimizing multiple queries on scientific datasets with partial replicas. In: GRID, pp. 259–266. IEEE (2007)

Demonic Testing of Concurrent Programs

Scott West, Sebastian Nanz, and Bertrand Meyer

ETH Zürich, Switzerland
firstname.lastname@inf.ethz.ch

Abstract. Testing presents a daunting challenge for concurrent programs, as non-deterministic scheduling defeats reproducibility. The problem is even harder if, rather than testing entire systems, one tries to test individual components, for example to assess them for thread-safety. We present *demonic testing*, a technique combining the tangible results of unit testing with the rigour of formal rely-guarantee reasoning to provide deterministic unit testing for concurrent programs. Deterministic execution is provided by abstracting threads away via rely-guarantee reasoning, and replacing them with "demonic" sequences of interfering instructions that drive the program to break invariants. Demonic testing reuses existing unit tests to drive the routine under test, using the execution to discover demonic interference. Programs carry contract-based rely-guarantee style specifications to express what sort of thread interference should be tolerated. Aiding the demonic testing technique is an interference synthesis tool we have implemented based on SMT solving. The technique is shown to find errors in contracted versions of several benchmark applications.

1 Introduction

The spread of multicore architectures has established concurrent programming as an increasingly indispensable part of software development, and causes an increasing need for suitable development tools. Of particular importance to industrial applications is support for debugging and testing of concurrent programs; such support is difficult to provide, however, because of the unpredictability and irreproducibility of thread scheduling, which makes interference between threads very difficult to discover. These difficulties have not stopped successful research into concurrent testing tools, e.g. [12,23,19,14,25]. The focus of this work is to address both the difficulty in finding concurrency bugs, while keeping the process reproducible and modular.

Modularity and reproducibility are, in particular, difficult challenges, as concurrent programs appear to be inherently non-modular and non-deterministic: independent threads carefully manipulate shared-data to work towards a common goal (multiplying a matrix, serving a web-page, etc.). To overcome these challenges, demonic testing as prescribed in this paper takes regular unit-testing of routines and contracts, such as preconditions, and uses them to determine whether a routine will fail in a concurrent setting. A high-level visualization can be seen in Figure 1, where the program-state is exported during testing to a

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 478–493, 2012.

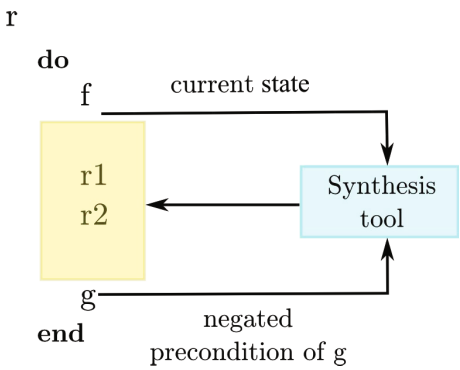


Fig. 1. Finding dynamic interference

constraint-based synthesis tool along with a precondition. If the synthesis tool can violate the precondition with the actions of another thread, this indicates that the routine is vulnerable to interference.

The overall process of demonic testing can be seen in Figure 2. Given a set of classes and one of their routines s chosen for testing, we perform:

- **Class-to-domain transformation**: collect all supplier classes for the routine s, and convert them to an abstract domain description for synthesis tool.
- **Routine instrumentation**: instrument the routine s to serialize the state.

These two steps produce a *domain description* and an *instrumented routine* which are passed to the remaining two modules of the system:

- **Testing tool** This component runs the instrumented version of s with relevant test cases. Test cases may for example be obtained from an automatic testing tool, such as AutoTest [18].
- **DemonL (synthesis) tool** This component takes the domain description as input and dynamic state of s recorded by the testing tool and produces sequences of interfering actions.

If the demonL tool finds interference for a test case, the interfering instructions are given. If no such interference can be found then the test succeeds.

An evaluation of the technique shows that it can successfully catch 7 out of 8 selected bugs of the concurrency bug collection [6], which include known bugs in major applications, such as Apache and MySQL – entirely without threads. The implementation of the technique is available [11,7].

The remainder of this paper is structured as follows. In Section 2 an overview of the approach and a running example are introduced. The overall technique is described in detail in Section 3, including the foundational concepts and the transformation of classes into the language of the demonL. We evaluate the technique in Section 4. Discussion on related work follows in Section 5, and we conclude in Section 6.

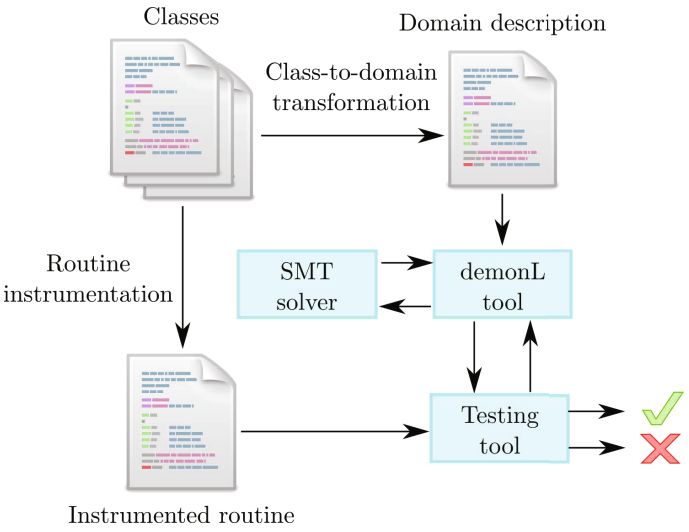


Fig. 2. Overview of the system architecture

2 Example Testing Run

To provide intuition for the demonic testing technique, this section introduces the running example.

The technique works with any language that can carry contracts, including C# (.NET code contracts) [5], Java (JML) [15], D, and Eiffel [17]; in the following we use the Eiffel syntax. Routine contracts in Eiffel are specifications in the form of pre- and postconditions as part of `require` and `ensure` clauses, respectively. The `old` keyword indicates that the value following it will be considered from the pre-state of the routine execution.

To apply the technique to non-Eiffel programs, a first step of translation into Eiffel is currently required. This requirement is a property of our current setup, not an intrinsic limitation of the method of demonic testing introduced in this article, which can be applied to any language supporting contracts.

Example. The Eiffel class `IDLE_COUNTER` in Figure 3 represents a collection of idle workers. The idle-counter can increase and decrease the number of idle workers. The `wait_for_idle` routine will decrease the number of idle workers if there are any, otherwise it waits on a condition variable until there are more idle threads. The objective is to test the routine `wait_for_idle` for usage in a concurrent setting.

Demonic testing uses the dynamic state at a given program point to analyze statically whether concurrent interference at that point could cause a fault, see Figure 1. We consider a *fault* to exist if the next instruction to be executed could have its precondition violated due to other threads modifying shared state. Part of the response from the static analysis is a sequence of instructions that would

```
class IDLE_COUNTER
feature
  num_idlers: INTEGER
  increment
    do ... end
  decrement
    require
      non_zero: num_idlers > 0
    do ...
    ensure
      num_idlers = old num_idlers - 1
    end
end
```

```
wait_for_idle
  do
    if num_idlers = 0 then
      mutex.lock
      if num_idlers = 0 then
        -- release locks on mutex
        -- and wait on condition
        non_zero.wait (mutex)
      end
      mutex.unlock
    end
    decrement
  end
```

Fig. 3. Work distribution example

move the program into a state that would cause a fault. These instructions represent other threads that may give rise to a failure in the program.

Example. While running a unit-test of the `wait_for_idle` routine in Figure 3, the tool instruments the routine `wait_for_idle` with calls to the synthesis tool. For a given test case, the tool reports whether or not interference could be found that will lead to a failure of the routine. In this example, the tool reports that an extra call to `decrement` immediately before the existing call to `decrement` will cause a violation.

There are two ways to respond to a warning by this method: either to modify the behaviour of the program so that it is not vulnerable to this kind of interference, or to refine the specification and only allow certain interference. In the case of Figure 3, synchronization instructions could be introduced to prevent concurrent access of the shared data.

Example. The developer can express that the interference found in Figure 3 does not occur by limiting the interference that can be generated. For example, the restriction could be: `num_idlers >= old num_idlers`. This specification disallows the environment from removing idle workers; upon retesting no violations are reported.

3 Demonic testing

This section presents the founding concepts and implementation of demonic testing as well as the approach to handling common synchronization primitives in a thread-free and modular way.

Demonic testing takes classes annotated with traditional contracts and rely-specifications and uses static analysis in combination with the state from runtime to indicate where there may be errors due to concurrent executions.

3.1 Application of Rely-Guarantee Reasoning

The rely-guarantee formalism [13] provides a framework to express and reason about interference in concurrent programs. The interaction between a component and its environment is included in the component's specification, allowing compositional reasoning about concurrent programs.

The formalism proposes an extension of the usual Hoare logic specification (P, Q) of a routine s with precondition P and postcondition Q, to a four-tuple (P, R, G, Q) which additionally contains a *rely*-condition R and a *guarantee*-condition G. The new conditions are binary predicates on states and describe the state changes that the environment (other threads) is allowed to make. A routine s satisfies its specification if, starting in a state satisfying P, under environmental interference adhering to R, s only makes state changes allowed by G, and finishes in a state satisfying Q.

The demonic testing approach uses a subset of the rely-guarantee concepts, namely the *rely*-conditions and the notion of *stability*, to specify interference generation for concurrent programs. The rely-specifications are manually added to the method under test, expressed as a postcondition with the tag `rely`. The `rely` tag indicates that this is only for demonic testing.

The concept of stability allows us to ascertain whether a routine can operate correctly in spite of the interference described by the rely-specification. Formally, the stability of a state-predicate p with respect to a rely-condition R is given as:

$$stable(p, R) \equiv \forall \sigma, \sigma'.\ p(\sigma) \wedge R(\sigma, \sigma') \rightarrow p(\sigma')$$

With the notion of the rely-condition one can express the goal of the testing strategy in the following terms: given the rely-condition R of a routine s under test, try to create interference that would drive the program to violate the precondition *pre* of some call in the body of s.

Example. In the running example, we have the following stability formula for the precondition of decrement:

$$num\_idlers(\sigma(this)) > 0 \wedge num\_idlers(\sigma'(this)) \geq num\_idlers(\sigma(this)) \rightarrow num\_idlers(\sigma'(this)) > 0$$

Demonic testing distinguishes itself from other techniques of program verification by the usage of a dynamic program state to reduce the need for program specification. The goal given to the demonL tool is merely the negation of the *stable* predicate, $\exists \sigma, \sigma'.\ p(\sigma) \wedge R(\sigma, \sigma') \wedge \neg p(\sigma')$. In demonic testing, this is formula simplified by two assumptions:

1. that the routine is correct without interference, and
2. that the test-cases driving the routine constitute the inputs on which it is expected to work correctly.

The first point allows us to assume $p(\sigma)$, the second allows us to remove σ as a quantified expression, as it is given by the dynamic state. This leaves solving only

$\exists\sigma'.\ R_\sigma(\sigma') \wedge \neg p(\sigma')$, where R_σ is the rely condition specialized to the concrete program state. Since the rely condition is specialized, it doesn't have to handle cases that never arise in normal program execution; this lowers the amount of required annotation. Additionally, there is no specification required for typically difficult to specify cases, such as loop variants and invariants.

3.2 The Domain Description Language

Program synthesis constructs a program that satisfies a given specification. Demonic testing uses program synthesis to construct interference, actions performed by other threads, which indicates errors in concurrent programs.

Facilitating the demonic testing approach are a language and tool: *demonL*. In the same spirit as the verification language Boogie [2], demonL serves as an intermediate language to express the allowable types of interference. The input to the tool consists of two parts: a domain and a goal.

The domain language is as follows:

$$
\begin{array}{ll}
\mathit{Domain} & ::= [\mathit{TypeDecl} \mid \mathit{ProcDecl}]^* \\
\mathit{TypeDecl} & ::= \textbf{type}\ \mathit{ident}\ \{\mathit{Decl}^*\} \\
\mathit{Decl} & ::= \mathit{ident} : \mathit{ident} \\
\mathit{ProcDecl} & ::= \mathit{ident}(\mathit{Decl}^*)[: \mathit{ident}]?\ \mathit{Pre}?\ \mathit{Post}? \\
\mathit{Pre} & ::= \textbf{require}\ \mathit{TaggedExpr}^* \\
\mathit{Post} & ::= \textbf{ensure}\ \mathit{TaggedExpr}^* \\
\mathit{TaggedExpr} & ::= \mathit{tag} : \mathit{Expr} \\
\mathit{Expr} & ::= \mathit{op}\ \mathit{Expr} \mid \mathit{Expr}\ \mathit{op}\ \mathit{Expr} \mid \mathit{Call} \\
\mathit{Call} & ::= \mathit{ident}(\mathit{Expr}^*)
\end{array}
$$

where *op* can be the common infix and prefix operators, with the addition of an `old` prefix operator.

To specify the desired initial and final states the following goal language is used, sharing the same expression and declaration syntax as the domain format.

$$
\begin{array}{ll}
\mathit{Goal} & ::= \mathit{Decl}^*\ \mathit{InitialState}\ \mathit{FinalState} \\
\mathit{InitialState} & ::= \textbf{initial}\ \mathit{Expr}^* \\
\mathit{FinalState} & ::= \textbf{final}\ \mathit{Expr}^*
\end{array}
$$

```
type Idle_Counter {num_idlers: Integer}
increment (this: Idle_Counter)
  ...
decrement (this: Idle_Counter)
  require
    non_zero: this.num_idlers > 0
  ensure
    this.num_idlers = old this.num_idlers − 1
```

Domain specification

```
this: Idle_Counter

initial
  not (this = null) and
  this.num_idlers = 1

final
  not (this.num_idlers > 0)
```

Goal specification

Fig. 4. The `IDLE_COUNTER` class in demonL

Example. Figure 4 shows a program written in demonL, corresponding to the class `IDLE_COUNTER` in Figure 3.

The domain describes the state through data structures, functions on the state, as well as procedures that transform the state. Procedures and functions are described with pre- and postconditions. The goal describes the entities in the system and constraints on the initial state and final state. The final state relates the initial and goal states through the use of `old` operator, which references the values in the initial state.

DemonL constructs an initial state that satisfies the initial constraints, a series of actions, and a final state that is the result of the actions applied in order, and also satisfies the final-state constraints.

Example. To find the possible interference that could be used to destabilize Figure 3, the goal specification found in Figure 4 is used. The goal specification contains the negation of the precondition of the `decrement` operation, here: `this.num_idlers <= 0`. However, if the goal includes the rely-condition restricting the interference to only non-decreasing effects on the number of idle workers, then the program is correct under the rely assumption.

```
final
  this.num_idlers >= old this.num_idlers and
  not (this.num_idlers > 0)
```

Again we can see the same shape stability criterion, $\exists \sigma'.\ R_\sigma(\sigma') \wedge \neg p(\sigma')$.

3.3 Class Transformation

We assume an input (Eiffel) class C has three components: C_{name}, C_{attrs}, and C_{routines}. C_{name} denotes the name of the class. The attributes of the class, C_{attrs}, are denoted by $a : t$ to indicate an attribute a that has type t. Every routine s in C_{routines} has a name, denoted by s_{name}. Also, every routine can have a pre- and postcondition, denoted by s_{pre} and s_{post}.

The translation function to convert class files into demonL domains (see Figure 2) is shown in Table 1. Note that the presentation of this translation function uses a pattern-matching style, with the function matching arguments in a top-down fashion.

- Attributes, along with the class name, are transformed into a datatype in demonL.
- Routines are transformed using *feat* directly into demonL procedures with pre- and postconditions.

The result of a function is denoted by having equality on the `Result` value, for example `Result = 2 * x`. Argument-list transformation of routines and functions explicitly includes the normally implicit self-reference in object-oriented programs. The translation of expressions is largely straightforward, with the target of a call moving to the first argument of the call, to coincide with the argument-list transformation.

Table 1. Translation function

$$
\begin{array}{rl}
trans(C) = & \{feat(C, f) \mid f \in C_{\mathsf{features}}\} \cup \{data(C)\} \\
\hline
data(C) = & \textbf{type}\ C_{\mathsf{name}}\ \{\ C_{\mathsf{attrs}}\ \} \\
feat(C, f) = & f_{\mathsf{name}}(args(C, f_{\mathsf{args}})) \\
& \quad \textbf{require}\ expr(f_{\mathsf{args}}, f_{\mathsf{pre}}) \\
& \quad \textbf{ensure}\ expr(f_{\mathsf{args}}, f_{\mathsf{post}}) \\
\hline
args(C, \boldsymbol{as}) = & (this : C_{\mathsf{name}}) :: \boldsymbol{as} \\
\hline
expr(\mathsf{args}, x.f(\boldsymbol{as})) = & f(expr(\mathsf{args}, x), expr(\mathsf{args}, \boldsymbol{as})) \\
expr(\mathsf{args}, e_1\ \mathsf{op}\ e_2) = & expr(\mathsf{args}, e_1)\ \mathsf{op}\ expr(\mathsf{args}, e_2) \\
expr(\mathsf{args}, \mathsf{op}\ e) = & \mathsf{op}\ expr(\mathsf{args}, e) \\
expr(\mathsf{args}, v) = & \begin{cases} v & \text{if } v \in \mathsf{args} \\ this.v & \text{otherwise} \end{cases}
\end{array}
$$

Example. An example of this translation process can be seen by examining how Figure 3 is translated to Figure 4.

3.4 Routine Instrumentation

As part of the technique, the routine under test must be instrumented (see Figure 2). The instrumentation augments the program execution so it is able to encode the dynamic state of the program for demonL. This procedure is straight-forward.

3.5 The demonL Tool

The output of the tool is the sequence of actions, and their arguments, that bring the program from the initial to the final state. Given the specifications in Figure 4, this would be a call to `decrement`. If the underlying SMT solver reports that the constraints are unsatisfiable, this indicates that no sequence of actions could be found. To avoid long synthesis times, the tool constructs sequences bounded by number of instructions and number of unique references for each user-constructed type.

However, because of the constraint-based nature of the encoding, first the tool solves the interference problem in a single step with *no* actions to constrain the transformation. This is equivalent to a *proof* of instability. If the tool determines that interference is possible then it tries to obtain the sequence of actions. If, even without constraints, it cannot find interference then interference is impossible no matter the actions or bounds given to the tool. This means that demonL's determination of the absence of interference is not limited by the inability of the tool to construct a sequence of appropriate instructions.

Having an intermediate language and tool offers substantial advantages to the application of the demonic technique: separating the complexity of encoding the verification conditions from the task of routine instrumentation, and the possibility to target more than one source language and and more than a single

solver in the back-end. The current technology choices for demonL are Eiffel as a source language to be translated to demonL, and Yices [9] as the SMT solver.

DemonL is similar to planning tools. In particular it allows the movement from an initial state to a final state by a series of actions. However, the specification of the initial state and the actions are permitted to be weaker than generally allowed by planning tools that use languages such as the Planning Domain Definition Language (PDDL) [16]. Where PDDL only allows the effect of an action to be expressed using certain atomic-terms our tool has no such restriction: any expression can be used to describe the effect of an action. For example, where a PDDL domain would require a post-condition such as `attribute = 5`, demonL is able to deal with with post-conditions such as `attribute > 3`. DemonL also does not assume determinism of the actions. These qualities are important when representing program specification, which are typically incomplete.

DemonL is available for download from [7].

3.6 Handling Synchronization Primitives

The use of threads to construct concurrent programs inherently exhibits two types of effects:

- the *necessary*, where a thread contributes a result to another thread, and
- the *incidental*, which are side-effects of necessary actions, and are also modifications to shared state.

When we consider concurrent applications as a combination of necessary and incidental effects, the necessary aspect of concurrency can be seen as a dependency, and the incidental aspect can be seen as interference. One thread depends on another to provide a computational result in a shared memory location. In threaded programs, these dependencies are made explicit by a mutex's `lock`, or a condition variable's `wait` routine.

When unit-testing a class or method, it is common to provide stub methods or objects in the place of dependencies. For example, a full database connection may be replaced with one containing only a small fixed selection of data.

Although mutexes, semaphores, and condition variables carry no explicit invariants, their usage in programs is almost always accompanied by an implicit invariant related to a resource. Consequently, they can have meaningful post-conditions that we can use to create stubs to test concurrent programs without requiring threads. They merely need to be replaced with normal function calls that ensure the same postcondition.

Example. Assume a simple producer/consumer-style program, such as that given in Figure 5. The call to `cond_var.signal` in the `produce` routine has the pre-condition that the number of products is greater than zero. The counterpart in the `consume` routine, the call to `cond_var.wait`, has the same post-condition: `product > 0`.

To create a stub for the call to `cond_var.wait`, replace the implementation of `wait` on the condition variable with

```
produce
  do
    product := product + 1
    if product = 1 then
      cond_var.signal
    end
  end
```

```
consume
  do
    if product = 0 then
      cond_var.wait
    end
    product := product - 1
  end
```

Fig. 5. Producer/consumer coordination

```
wait do product := product + 1 end
```

The new `wait` satisfies the invariant for the condition variable, and requires no other thread to work. The corresponding stub for `signal` would similarly have `product > 0` as a precondition and an empty body.

4 Experimental Evaluation

It is essential for a testing technique to be judged by its reaction to bugs that occur in real software. For this purpose, we use a selection of bugs from a concurrency bug database [26,6] to determine if demonic testing can detect and help form fixes for the faults. No particular criteria was used to select bugs from the database, besides striving for an overall diversity of faults. All experiments were carried out on an Intel Q6600 2.4GHz with 4GB of RAM.

4.1 Conversion from Source Programs

All of our test cases are extracted from real projects and translated into Eiffel. Since well-known concurrent applications with specifications are rare, we slice the non-essential elements from well-known code then convert it to Eiffel and add contracts. This is also done to enable the analysis of bugs from many languages, while minimizing the differences due to language features. To see an example of this process, the original Apache C-code for the running example is given in Figure 6. The main differences are the removal of the recycled pool functionality, and the removal of the explicit return-value checking of concurrency primitive (locks, condition variable) operations that is typically handled by exceptions in languages that support them.

4.2 Results

Table 2 lists the collection of concurrency bugs that we use to perform our evaluation; the first seven are from the bug database, with the last being a well-known Java standard library bug. All bugs have been replicated using the demonic testing technique, with the exception of MySQL #169, as explained in the discussion at the end of the section. Inspired by the AutoTest approach, work initially began using Eiffel as the source language; to broaden the scope of the evaluation we translated bugs from multiple other languages. These examples are available for download [7].

```
apr_status_t ap_queue_info_wait_for_idler
  (fd_queue_info_t *queue_info,
   apr_pool_t **recycled_pool)
{
  apr_status_t rv;
  *recycled_pool = NULL;
  if (queue_info->idlers == 0) {
    rv = apr_thread_mutex_lock(
      queue_info->idlers_mutex);
    if (rv != APR_SUCCESS) {
      return rv;
    }
    if (queue_info->idlers == 0) {
      rv = apr_thread_cond_wait(
        queue_info->wait_for_idler,
      queue_info->idlers_mutex);
      if (rv != APR_SUCCESS) {
        apr_status_t rv2;
        rv2 = apr_thread_mutex_unlock(
          queue_info->idlers_mutex);
        if (rv2 != APR_SUCCESS) {
          return rv2;
        }
        return rv;
      }
    }
    rv = apr_thread_mutex_unlock(
      queue_info->idlers_mutex);
    if (rv != APR_SUCCESS) {
      return rv;
    }
  }
  apr_atomic_dec32(&(queue_info->idlers));

  ... recycling of data structures
}
```

Fig. 6. Original `wait_for_idle` routine from Apache

Table 2. Bug collection

	Program/Bug	Bug type	LOC	Annotation			Time (s)
				Lock	Simple	Complex	
1	Apache #21285	Atomicity violation	125	0	4	0	0.982
2	Apache #25520	Data-race	101	0	2	0	0.124
3	Apache #45605	Data-race	227	1	4	0	0.217
4	MySQL #169	Atomicity violation	69	–	–	–	–
5	MySQL #644	Data-race	124	0	3	1	0.939
6	MySQL #791	Data-race	113	0	1	0	0.139
7	pBZip2	Order violation	168	1	1	0	2.289
8	Java Vector	Data-race	70	0	2	0	0.032

The time taken to generate interference, or determine that none exists, was measured for the bugs that were successfully tested. The average time taken was 100ms for each request to demonL. This time is different from the times in Table 2, as each time in the table may include many requests to demonL.

The rest of this section analyzes the effort required to write the necessary program contracts, the conversion process, and concludes with a discussion of the notable properties of the technique.

4.3 Annotation Complexity

In any approach which requires the addition of specification via program annotation, the burden that this annotation places on the programmer is highly relevant. Although difficult to measure objectively, we place the annotations into three categories:

– Lock – a rely-annotation denoting that a lock protects some shared data from change by another thread.

- Simple – a non-concurrency-related program annotation stating a property of the program that is either a non-null check for a reference, or a linear equation.
- Complex – a non-linear expression, or a frame condition that is necessary to limit the scope of an operation.

Table 2 collects the types of annotations required in the test cases. These are the types of annotations required for demonic testing to give the correct cause of the bug in the full program in the cases of Apache, MySQL, and pBZip2. In the Java vector implementation one of many possible causes is given, as it is part of a library.

4.4 Discussion

The Apache bug #45605 example is notable due to the the double-check present in the `wait_for_idle` routine. Separate tools exist to classify some data-races as "potentially benign" [20]; the double-check pattern is benign and difficult for pure data-race checkers to deal with. Demonic testing does not require any secondary approaches to accomplish this: the determination of benign vs. malignant data-races is based on the program contracts.

The only bug from our test set which could not be discovered using demonic testing came from MySQL bug #169. The reason is that the invariant of the program could not be expressed without either ghost variables or artificially adding more data.

Incorrectly stated rely conditions will lead to both false-positives and false-negatives, as these essentially form an axiom of the routine to which they belong. However, as in Table 2, all rely conditions required were of a very plain type, merely indicating that a certain lock protects some shared state.

The bounded synthesis done by demonL may affect the results by not considering interference from sequences of instructions that exceed the bound. However, this bound concerns the search for instruction sequences; there is also an initial unbounded-verification that demonL performs to determine the stability before trying to synthesize interference. The worst case is that the tool is unable to find the sequence of actions but still reports whether interference is possible. All bugs our evaluation examined required only single-action interference to become evident. This suggests that many concurrent bugs manifest themselves with little prompting; and that causing errors to present themselves in threaded executions is difficult due to scheduling rather than maintaining very complicated invariants.

This experimental method is limited by the number and selection of examples, it is possible that drawing on a larger pool of examples would offer greater insight into the properties of demonic testing. However, the small sample size is mitigated by the wide variety of types of concurrency bugs. Although every effort was made to make a faithful reproduction of the programs in the target language, there is the possibility transcription errors while moving between different programming languages.

5 Related Work

The idea of using routine specifications to discover concurrency errors is not unique to demonic testing. The Colt tool [24] for Java also uses this approach. However, their approach is less general, relying hard-coded specification of the existing Java concurrent collection classes. Demonic testing is more generic as it works with user-supplied classes and specification, and as well allows finer-grained control of what constitutes an error through the usage of rely-conditions.

A common practice for testing of concurrent programs is load or stress testing. This frequently proves to be ineffective as in typical testing environments interleavings might only change marginally from one test run to the other. To force different interleavings, Edelstein et al. [10] present the ConTest tool, which combines a technique for deterministic replay of concurrent programs [4] with a heuristic for varying thread schedules by seeding sleep calls at synchronization points in the program.

Dynamic model checking [12,19,25] provides a more systematic approach by systematically exploring all possible thread interleavings. The search is stateless in that it provides a specialized scheduler that runs the program in its real execution environment, and hence can avoid storing concrete program states. The main problem is to overcome state explosion, which makes brute-force exhaustive search infeasible for large applications. Techniques such as partial order reduction as employed in the VeriSoft tool (Godefroid [12]) or preemption bounding (giving priority to schedules with fewer preemptions) in the CHESS tool (Musuvathi et al. [19]) can mitigate the effects of state space explosion only to a small degree. Wang et al. [25] propose a heuristic where ordering constraints learned from successful runs are used to guide the selection of interleavings for future runs. All of the above works focus on varying thread interleavings to produce undesired behaviour. Demonic testing differs from this approach by considering the routines in a program and finding sequences which lead to the violation of a program invariant, avoiding an exhaustive search of interleavings.

A number of works use combinations of dynamic and symbolic analyses to improve testing of concurrent programs. Sen and Agha [23] use a combination of concrete and symbolic execution, termed concolic execution, to test multi-threaded Java programs with the tool jCUTE. Symbolic execution produces input values that guide the concrete execution to alternate paths; concrete execution guides the symbolic computation along a concrete path to concretize any values that cannot be handled by a constraint solver. Besides producing alternate input values, their technique also systematically generates thread schedule variations such that potentially all causal structures of a concurrent program can be explored. Sen [22] introduces RaceFuzzer, an algorithm which uses race warnings from race detection tools to create problematic interleavings during testing in order to eliminate false positives automatically. Park et al. [21] propose CTrigger, a testing tool to expose atomicity violation bugs. The tool analyzes traces to find unserializable interleavings then these interleavings are explored during testing to expose bugs. Kundu, Ganai, and Wang [14] present a framework that combines conventional testing with symbolic analysis. A test harness invokes the

program with random test values. Concrete traces are relaxed into concurrent trace programs, which capture all linearizations of events that respect the control flow of the program. The concurrent trace programs are then symbolically verified. All these techniques combine in some way the dynamic execution of programs with symbolic computation and verification, and most closely resemble the work presented in this paper. However, they, like all other related work shown, are not able to achieve truly modular testing of concurrent software; they all depend on multithreaded executions or traces.

Contracts have been used successfully in unit testing of sequential software [18], where they can provide test oracles and filter inputs for random testing. Araujo et al. [1] evaluate the use of contracts in a concurrent setting, based on an extension of the JML [15] contract semantics. They found contracts as test oracles effective in finding and diagnosing concurrency-related faults on an industrial case study in Java/JML. In contrast to this work, demonic testing emphasizes the use of contracts also for symbolic analyses, in addition to test oracles.

Rely-guarantee reasoning has been applied in testing of concurrent programs. Dingel [8] uses the state exploration tool VeriSoft [12] for rely-guarantee verification of C/C++ components. The component code is executed in parallel with an environment which generates initial states, monitors the component execution, and generates responses. If a program step is found to violate one of the guarantees, a flaw is found. Blundell et al. [3] use labelled transition systems to model the behaviour of components, whereas demonic testing works directly on source code. Assumptions on the model-level are used as environments in which individual components are executed. The execution results in traces which are in turn checked against the guarantees of the model. Failure of a check suggests an incompatibility between a model and its implementation.

6 Conclusion

Until recently, the testing of concurrent systems has generally been regarded as inferior to static approaches. The realization that purely static reasoning also faces problems of scalability or precision when applied to concurrent systems has led to a more pragmatic assessment, leaving testing its due place, as evidenced by the approaches reviewed in the previous section.

Unlike many of these approaches, which are only suitable for testing entire systems, demonic testing can be applied to the important problem of unit testing for concurrent programs. Through its combination of dynamic and symbolic techniques, demonic testing provides two significant benefits over other proposals. First, it leverages available testing tools for sequential programs, which it uses as an essential part of its architecture. Second, instead of searching the state space of thread interleavings, demonic testing uses program synthesis as a constructive means to find problematic thread interference. If a test fails, a test case and a problematic sequence of interactions is available for analysis.

Acknowledgments. The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC Grant agreement no. 291389, the Hasler Foundation, and ETH (ETHIIRA). Earlier work has also benefited from grants from the Swiss National Foundation and Microsoft (Multicore award).

References

1. Araujo, W., Briand, L., Labiche, Y.: On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In: Proc. ESEM 2011. IEEE Computer Society (2011)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Blundell, C., Giannakopoulou, D., Păsăreanu, C.S.: Assume-guarantee testing. In: Proc. SAVCBS 2005. ACM (2005)
4. Choi, J.-D., Srinivasan, H.: Deterministic replay of Java multithreaded applications. In: Proc. SPDT 1998, pp. 48–59. ACM (1998)
5. Code contracts (2011), `http://research.microsoft.com/en-us/projects/contracts/`
6. Collection of Concurrency Bugs (2011), `http://www.eecs.umich.edu/~jieyu/bugs.html`
7. Demonic test case downloads (2011), `http://se.inf.ethz.ch/people/west/demonic-cases/`
8. Dingel, J.: Computer-assisted assume/guarantee reasoning with VeriSoft. In: Proc. ICSE 2003, pp. 138–148. IEEE Computer Society (2003)
9. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
10. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for testing multi-threaded Java programs. Concurrency and Computation: Practice and Experience 15(3-5), 485–499 (2003)
11. EVE project (2011), `https://svn.eiffel.com/eiffelstudio/branches/eth/eve/`
12. Godefroid, P.: Model checking for programming languages using VeriSoft. In: Proc. POPL 1997, pp. 174–186. ACM (1997)
13. Jones, C.B.: Development Methods for Computer Programs including a Notion of Interference. PhD thesis, Oxford University (June 1981)
14. Kundu, S., Ganai, M.K., Wang, C.: Contessa: Concurrency Testing Augmented with Symbolic Analysis. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 127–131. Springer, Heidelberg (2010)
15. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Software Engineering Notes 31, 1–38 (2006)
16. McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL: The planning domain definition language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control (1998)
17. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall (1997)

18. Meyer, B., Fiva, A., Ciupa, I., Leitner, A., Wei, Y., Stapf, E.: Programs that test themselves. IEEE Computer 42, 46–55 (2009)
19. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and reproducing Heisenbugs in concurrent programs. In: Proc. OSDI 2008, pp. 267–280. USENIX Association (2008)
20. Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., Calder, B.: Automatically classifying benign and harmful data racesallusing replay analysis. ACM SIGPLAN Notices 42(6), 22–31 (2007)
21. Park, S., Lu, S., Zhou, Y.: CTrigger: Exposing atomicity violation bugs from their hiding places. In: Proc. ASPLOS 2009, pp. 25–36. ACM (2009)
22. Sen, K.: Race directed random testing of concurrent programs. In: Proc. PLDI 2008, pp. 11–21. ACM (2008)
23. Sen, K., Agha, G.: CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 419–423. Springer, Heidelberg (2006)
24. Shacham, O., Bronson, N.G., Aiken, A., Sagiv, M., Vechev, M.T., Yahav, E.: Testing atomicity of composed concurrent operations. In: Proc. OOPSLA 2011, pp. 51–64 (2011)
25. Wang, C., Said, M., Gupta, A.: Coverage guided systematic concurrency testing. In: Proc. ICSE 2011, pp. 221–230. ACM (2011)
26. Yu, J., Narayanasamy, S.: A case for an interleaving constrained shared-memory multi-processor. In: Proc. ICSA 2009, pp. 325–336. ACM (2009)

Towards Certified Runtime Verification

Jan Olaf Blech[1], Yliès Falcone[2], and Klaus Becker[1]

[1] fortiss GmbH, Munich, Germany
{blech,becker}@fortiss.org
[2] Université Grenoble I, Grenoble, France
ylies.falcone@ujf-grenoble.fr

Abstract. Runtime verification (RV) is a successful technique to monitor system behavior at runtime and potentially take compensating actions in case of deviation from a specification. For the usage in safety critical systems the question of reliability of RV components arises since in existing approaches RV components are not verified and may themselves be erroneous.

In this paper, we present work towards a framework for certified RV components. We present a solution for implementations of transition functions of RV monitors and prove them correct using the Coq proof assistant. We extract certified executable OCaml code and use it inside RV monitors. We investigate an application scenario in the domain of automotive embedded systems and present performance evaluation for some monitored properties.

1 Introduction

Behavioral guarantees are an important prerequisite for using embedded systems in safety critical environments. Runtime verification [HG05, PZ06, FFM09, BH11] (RV) has become an important technique to monitor a system's behavior at runtime and take compensating actions in case of deviation from a specification. In RV, a system is typically extended with instrumentation code that communicates with a monitor. The monitor may be realized as an external program, the monitor is then referred as an outlined monitor. Once an abnormal behavior is detected, the monitor tries to bring the system into a fail-safe state using some feedback loop. This increases the confidence to handle system errors appropriately when the system is running, and, helps discovering them during testing.

Going one step beyond classical RV: for achieving an even higher level of confidence the question of whether an RV system itself has been implemented correctly arises. We address this question in this paper. In particular we guarantee that runtime-monitors do indeed monitor the desired specification and show the practicability of these runtime monitors for regular properties expressed with regular expressions.

The described approach targets OCaml based runtime monitors and their verification using the Coq proof assistant [Coq12]. Coq based runtime monitors

T. Aoki and K. Tagushi (Eds.): ICFEM 2012, LNCS 7635, pp. 494–509, 2012.

can be extracted as OCaml code and verified in the Coq environment. In the context of this paper, the code that is verified and extracted out of Coq is said to be *certified*.

Our approach is suitable for *regular expression based properties* in the embedded systems domain. As an analyzed and implemented example we are discussing properties that can be monitored while analyzing the traffic on a bus structure. This paper aims to demonstrate that *certified runtime verification* is feasible and can be used in safety-critical application domains. More precisely, this paper features the following contributions:

- Verified runtime monitors automatically generated out of Coq and a method to verify regular expression based monitors.
- An OCaml based framework for runtime monitoring and its evaluation.
- An experimental application for monitoring properties inspired by needs from the automotive embedded systems domain. Furthermore, we present a description of the ingredients of a certified RV system and dependencies between different RV components.

The main focus of the experimental application shall demonstrate the possibility to integrate the approach for real applications in the embedded systems domain especially in the automotive area. We are not at the stage of deploying a concrete application in a car. Many parameters are not fixed yet, e.g., the embedded devices and the exact type of bus. For this reason we evaluate some aspects of the approach using standard hardware in this paper, but mention the constraints for real applications in the automotive area.

Paper Organization. This paper is structured as follows. Related approaches are discussed in Section 2. Section 3 introduces guiding examples from the automotive domain that are regarded throughout this paper. Prerequisites including definitions on RV correctness are described in Section 4. Our OCaml based runtime monitors are introduced in Section 5. Section 6 describes runtime monitor formalization, and correctness proofs. An evaluation including a study on the integration of our Coq based code is presented in Section 7 and Section 8 features a conclusion and ideas for future work.

2 Related Approaches

Runtime Verification principles. Over more than a decade, the field of *runtime verification* has produced many frameworks dedicated to the verification of the behavior of monolithic programs w.r.t. user-defined specifications. From an abstract point of view, most of these approaches proceed as follows. Two inputs are needed: a user-defined specification characterizing some desired or proscribed behavior, and an implementation to be runtime-checked. The (abstract) events of this specification are related to the (concrete) events of the program. An instrumentation technique is then used to observe the execution of these events when the program is executed. The so-called monitor is generated from the specification. A monitor is a decision procedure for the specification: it is fed by events coming from the program and indicates specification fulfillment or violation.

RV implementations. Many tools have been proposed as implementations of the existing runtime verification frameworks. A large effort is the Java-MOP line of work conducted by Rosu et al. (see [MJG+11] for an overview). Java-MOP uses as input specifications which can be written in different formalisms (e.g., LTL, regular expression, context-free grammars). Java-MOP allows to generate an AspectJ aspect that instruments the underlying program (using weaving) and embeds the (automatically generated) monitor. Besides its genericity, Java-MOP is also efficient as demonstrated by experimentation.

On the other hand, a series of tools and approaches are based on the (less efficient) paradigm of rewriting. These approaches focus on expressiveness of the specification formalism (rather than the runtime efficiency). A major effort in this regard is the endeavor conducted by Barringer and Havelund with the tools Eagle [BGHS04], RuleR [BGHS04, BRH10], LogScope [BGHS10], and TraceContract [BH11]. Eagle handles LTL formulae as input and uses the techniques of progression that was proposed earlier in planning. RuleR is a more general system where specifications are directly encoded as a set of rewrite rules. This confers RuleR the ability to handle very expressive specifications. From an abstract point of view, LogScope can be seen as a variant of RuleR internally using state-machines. TraceContract is an embedding of LogScope in the Scala programming language (as an internal domain-specific language).

In addition to these two major research endeavors there are the tools TraceMatches [AAC+05], JLO [SB06], and LARVA [CPS09]. TraceMatches is an extension of AspectJ allowing to write regular expressions over pointcuts. JLO allows to generate monitors from LTL formulae where events are AspectJ pointcuts. Finally, LARVA allows to monitor different specification formalisms such as Lustre and duration calculus. LARVA translates specifications into the so called dynamic event timed automata and then uses AspectJ to weave the monitor.

Bridging the gap between formalized requirements, monitor specifications and assuring these properties for an implementation at runtime is addressed in the MaC framework [LKK+99, KVL+99].

Formal Treatment. As for the formal verification part of the work, a summary of usages of RV for certification has been proposed by Rushby [Rus08]. Additional ideas for monitoring systems in the context of certification are stated in [SH11]. Up to our knowledge, we are the first who have actually realized and evaluated certified RV monitors. Moreover, none of the previous tools or framework contains a certified subset. The presented verification technique, explicitly establishing a simulation relation that captures characteristics between property and states of a monitor in Coq, is similar to one of the authors work on compiler verification [BG11].

3 Guiding Examples

Figure 1 presents an abstract view on a guiding example from the embedded automotive system domain. There are sensors, a control unit and an actuator

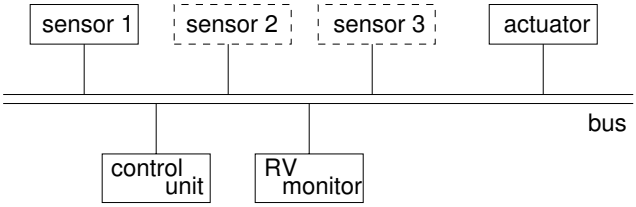


Fig. 1. An example bus

connected to a bus with with some timing guarantees[1]. The control unit receives data from the sensors and processes them. Based on this, the actuator receives messages from the control unit. Sensor 1 is mandatory, while the other sensors are optional and can be added later.

The monitor observes the bus. In particular, we are interested whether the communication with actuator and sensor conforms to a given specified protocol. If the protocol is violated, the monitor notices this and can trigger a handling for this problem. However, the error-handling itself is not part of the monitor. Different kinds of errors due to hardware failures (e.g., bit-shifts, packet loss) or software errors can occur and need to be detected.

The bus features a time signal every 10 ms. The fact that reasonable data is sent on the bus by either a sensor, the actuator, or the control unit is abstracted into events.

In a real system, the first sensor might be a manual control providing a user interface for a functionality that is realized as an actor, while the two other sensors might analyze the environment to improve the service quality. With the advent of real-time operating systems that provide some timing and non-interference guarantees for parallel execution (e.g., PikeOS [KW]) it becomes possible to execute the control unit and the monitor in parallel even on the same processor core. The rate of arriving events is typically in the lower ms range, while an operating system's context switch can be typically done in a few μs.

A special feature of the given example is the ability to add and remove components to the IT system of a car during runtime, including connecting and disconnecting them from a bus. It is especially crucial that some core system behavior is preserved – and runtime monitored in this process. Two example applications and properties are regarded in the scope of this paper.

3.1 A Rain-Sensor Application

A Rain-Sensor senses whether it is raining and determines the rain intensity. It sends the calculated intensity value to a Wiper-Controller, which analyses the value and sends control values to a Wiper-Actuator. Communication is done using the bus. The communication between system components is shown in Fig. 2.

[1] cf. existing bus systems used in the automotive area with timing guarantees in the embedded domain: FlexRay [FRay05], TTEthernet [SKS10] or TTP [Kop93].

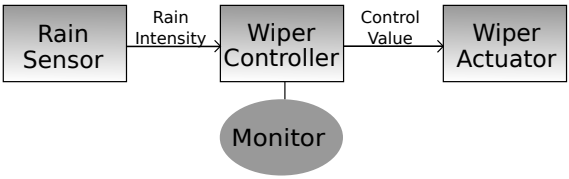


Fig. 2. Example application

For the given example we have realized the communication between sensor and controller in the following way: The Rain-Sensor encodes the rain intensity in 10 single event bits, followed by a parity event bit P. Each event can be either 0 or 1. The parity bit is included to be able to recognize if one bit of the event stream flips during transmission. The more $1s$ are sent, the higher is the rain intensity. After this sequence of 11 events has been sent, the next sequence is sent, transmitting a new value of rain intensity. The corresponding regular expression is: $1(1)^n(0)^m0P \quad$ such that $n + m = 8$ and $P \in \{0, 1\}$.

The regular expression is checked by a monitor. In addition to this, the monitor checks the parity bit P. If the number of previous $1s$ was odd (n was even) P shall be 1. In addition, the monitor can check another consistency condition that the rain-intensity (number of $1s$) in two succeeding sequences should only change by an amount of at most 2. This consistency condition reflects that the rain amount is unlikely to change much in the very short time interval of 10 ms. For instance, the sequence 11100000001 can be followed by 11111000001, but not by 11111111000.

Regarding the communication between controller and actuator: The controller counts the number of $1s$ (denoted $\#1$) received from the Sensor per sequence (without counting the parity) and sends a control value $C \in \{0, 1, 2\}$ to the actuator after each sequence. If $0 < \#1 \leq 2 : C := 0$; if $3 \leq \#1 \leq 6 : C := 1$; otherwise $C := 2$. The value of C denotes the speed of the wiper. For instance, if there is no rain $\#1 = 1$ and $C = 0$ denote that the wiper is off.

Due to the constraint that $\#1$ change at most by 2 in two succeeding sequences, the sequence of sent control values C is also constrained. After C was 0, C cannot become 2, but only 0 or 1. Assuming that $C = 0$ is the start and end state, the possible sequences of C can by expressed by $0^+(1^+(2^+1^+)^*0^+)^*$. This can be verified during runtime by a second monitor.

The monitors observe if both the sequence of received values by the controller and the sequence of sent control values are valid.

In addition to properties specified by regular expressions, here we also observe behavior that is not specified in the regular expression itself: we check the "maximal rain-intensity variation by two" property between two signals and the parity bit. Such properties are checked by our monitors and can be specified in Coq and generated out of the Coq specification. However, only the regular expression part is verified. It is important for us, that our framework supports monitors that are able to observe additional constraints – thus, these monitors are more restrictive than the regular expression. This enlarges the monitor but

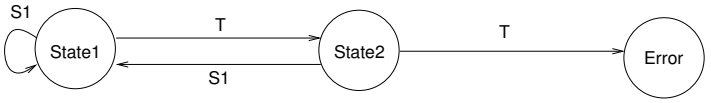


Fig. 3. Automaton for property 1

since these aspects may be less critical they do not have to be verified. In the given example, however, it would be possible to specify the extra constraints by large regular expressions.

3.2 Timing-Properties

Our bus systems typically features some time events. This is used to evaluate properties requiring that certain signals arrive in certain time intervals. The bus systems features the following events:

$$\Sigma = \{tick(T), actuator(A), sensor_1(S_1), sensor_2(S_2), sensor_3(S_3)\}$$

We assume the requirement that between two time ticks (10 ms interval) there is at least one $sensor_1$ event and the *actuator* status should be updated. Furthermore, either $sensor_2$ or $sensor_3$ should send a message event between two ticks.

The formalization of the previous requirement is ensured by the conjunction of the three following properties expressed with regular expressions:

1. $\phi_1 = (S_1 + T.S_1)^*.(T + \epsilon)$
2. $\phi_2 = (A + T.A)^*.(T + \epsilon)$
3. $\phi_3 = ((S_2 + S_3) + T.(S_2 + S_3))^*.(T + \epsilon)$

When abstracting from the other events, the first property can be realized by an automaton as depicted in Fig. 3.

4 Prerequisites

We now introduce prerequisites for our RV framework: a formalization of the notions of correctness of RV and the minimal concepts of regular expressions used to state properties.

4.1 A Formalization of RV Basics

In our RV scenario we distinguish a syntactical representation (e.g., the source code) of a system s, its instrumentation s^I and the syntactical representation of a monitor m. The instrumentation may modify the source code of a system.

- Assuming that an operational semantics can be assigned to s and that a *system state* of type Σ_s and *concrete events* of type Σ_c can be observed during runs associated with this semantics. The semantics of s is given by a function σ returning a set of traces in $(\Sigma_s \times \Sigma_c)^*$. The Σ_c events abstract system states. The motivation for distinguishing system states and concrete events is that system states represent all important information to determine the control flow and important actions of the system. Concrete events explicitly specify observable system behavior.
- The semantics of s^I is given by another function – for simplicity it is also denoted σ returning a set of traces. Each trace has the type:

$$(\Sigma_s \times \Sigma_c \times (\Sigma_a \cup \{ignore\}))^*$$

  Each trace element comprises a tuple of a system state, two corresponding events of system events in Σ_c and their instrumented counterparts in $\Sigma_a \cup \{ignore\}$. In the case of *ignore* no abstract corresponds to a concrete one. In the deployed RV system this is used to reduce communication overhead between the instrumented system and the monitor.
- Monitors are defined as state transition systems comprising monitor states m_{States} and a transition function $m_{Step} : (m_{States} \times \Sigma_a) \to (m_{States} \times bool)$ taking a monitor state and an abstract event and returning a new (updated) monitor state and a verdict.
  In addition to this, the monitor comprises code for communicating with the instrumented system and calling m_{Step}.
- The semantics of m with s^I running in parallel is denoted: $s^I||m$.
  The semantics is given by another function – for simplicity it is also denoted σ as a set of traces, each trace comprising tuples of four components. A trace has the type: $(\Sigma_s \times \Sigma_c \times (\Sigma_a \cup \{ignore\}) \times bool)^*$
  Traces comprise system states Σ_s, concrete events Σ_c, their instrumented counterparts $\Sigma_a \cup \{ignore\}$ and truth values returned by the monitor.

Traces of systems. Our correctness definitions use the projection of sets of traces (e.g., given by one of the σ functions) to the Σ_s^* parts (denoted T_s for a set of traces of tuples T). Furthermore, we need projections for the Σ_c^* and Σ_a^* (denoted T_c and T_a for a set of traces of tuples T). The event *ignore* is omitted in this projection. A projection for sets of the $bool^*$ parts of traces of tuples (denoted T_b for a set of tuples of traces T) is also needed.

The fact that a system s can generate a trace t ($t \in \sigma(s)$) is denoted $s(t)$. Likewise we introduce the notations $s^I(t)$ and $(s^I||m)(t)$ and projections $(\sigma(s^I))_c$ and $(\sigma(s^I||m))_c$ to restrict sets of traces to the Σ_c^* parts. The fact that a monitor m accepts a trace t_a is denoted $m(t_a)$. In the case of safety properties this means that its output trace contains only *true*.

Correctness of instrumentation. It is possible that the instrumentation or running the monitor in parallel with the instrumented system does change the semantics of the original system due to side effects. Thus, we require a definition

of correct instrumentation and monitor integration with respect to the uninstrumented system. This correctness definition is done by checking equality of sets of (projected) sets of traces.

Absence of side-effect of instrumentation is defined as:

$$\sigma(s)_s = \sigma(s^I)_s \wedge \sigma(s)_c = \sigma(s^I)_c$$

Furthermore, the instrumentation is responsible for abstracting concrete events. Abstract and concrete events shall correspond to each other using an abstraction function $\psi : \Sigma_c \to (\Sigma_a \cup \{ignore\})$. Thus, we require that applying ψ to the Σ_c component in a tuple in each trace is equal to the $\Sigma_a \cup \{ignore\}$ component.

Correctness of instrumentation is defined as the conjunction between absence of side-effects and the correspondence of concrete and abstract events.

Correctness of monitor integration Correctness of monitor integration requires a correct instrumentation and the following conditions:

- The monitor does not pose any side-effects on the instrumented system: $\sigma(s^I)_s = \sigma(s^I||m)_s \wedge \sigma(s^I)_c = \sigma(s^I||m)_c \wedge \sigma(s^I)_a = \sigma(s^I||m)_a$
- The monitor is not affected by side-effects: $\forall\ t\ \in (\Sigma_s \times \Sigma_c \times (\Sigma_a \cup \{ignore\}) \times bool)^*\ .\ \sigma(s^I||m)_b(t) \quad \text{iff} \quad m(t_a)$

Monitor correctness. Correctness of a monitor m_φ is defined with respect to a property φ. When a trace t fulfills a property φ, we note it $\varphi(t)$. Thus, monitor correctness is defined as: $\forall\ t_a\ \in \Sigma_a^*\ .\ \varphi(t_a) = m_\varphi(t_a)$

Combined correctness properties. An RV system $s^I||m_\varphi$ is considered correct with respect to a property φ iff: 1) the instrumentation is done correctly, 2) the monitor has been integrated correctly, and 3) the monitor is correct with respect to φ.

In addition, it has to be ensured that the system is a correct deployment and compilation of $s^I||m_\varphi$ which is not in the scope of this paper.

4.2 Regular Expressions

Regular expressions are defined as an inductive datatype Φ_{Σ_a} parameterized with a set of (abstract) events Σ_a. In our Coq formalization they comprise constructors for atoms, concatenations, disjunctions, the star and plus operators, and the epsilon (corresponds to an empty list of events) and empty expressions (corresponds to nothing). Furthermore, we have defined abbreviations to ease the specification like concatenating n-times the same (sub-)regular expression to each other.

Semantics of regular expressions. The semantics of a regular expression is defined in the usual way, by associating a regular expression ϕ with a function that indicates whether a list of events el is in the language described by ϕ. We note $\phi(el)$ when this is the case. We define an equivalence relation $\simeq$ over regular expressions ϕ, ϕ' that describe languages over the same vocabulary Σ:

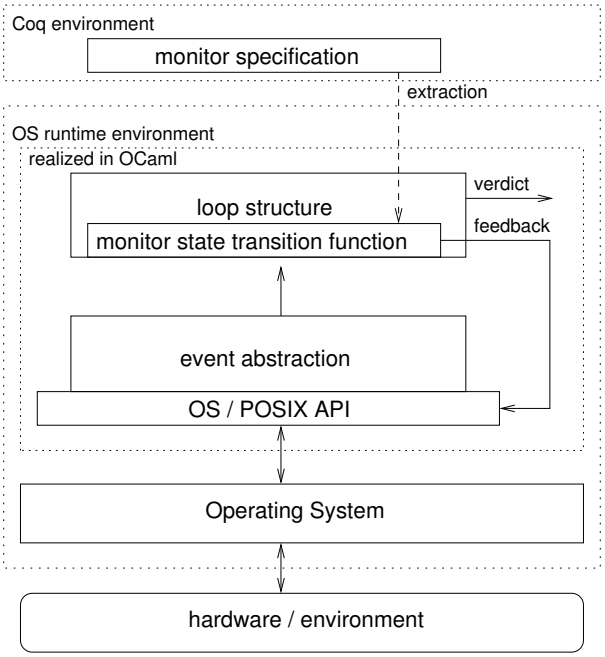


Fig. 4. Our RV monitoring environment

$$\forall\ el \in \Sigma^*.\ \phi(el) = \phi'(el) \text{ iff } \phi \simeq \phi'$$

This partitions syntactical representations of regular expressions into semantic equivalence classes.

Accepting an event. When comparing a regular expression ϕ to a list of events, it is helpful to define an operator that returns a modified regular expression that captures the effect of having consumed an event e. For this reason, we define the / operator that has the following property:

$$\forall\ e\ el\ .\ \phi(e :: el) \quad \text{iff} \quad \phi/e(el)$$

where $e :: el$ is the OCaml notation for the concatenation of an event e to a list of events el. Note that, for space reasons, we do not give the full formal definition specifying how this is achieved for regular expressions, but this is part of our Coq formalization.

5 OCaml Based RV Monitors

Our RV framework distinguishes between an instrumented system and monitors. Outlined monitors observe the system behavior by using information provided by the instrumented system.

Figure 4 shows the environment in which our RV monitors are deployed. RV monitors are relatively small pieces of software and are in this work realized as a state-transition function which performs transitions of the internal monitor state. This state-transition function is called from a loop structure over and over again, thereby consuming behavioral events from the system and emitting an overall system status value to indicate possible deviations from an acceptable system behavior. The events received by the RV monitor are originating from abstraction functions which observe, e.g., a stream from a Posix socket. This stream can contain concrete events which are originating from the instrumented system. Here, the monitor functionality is realized in OCaml. Communication with the rest of the system and the environment is done using the operating system API.

While we generate the state transition functions for our RV monitors out of Coq specifications which are verified in the Coq environment, the state transition functions have to be embedded into a monitor environment which takes care of the monitor's communication with the system, as shown in Fig. 4.

The OCaml part of the monitor is divided into three parts:

- A generic part, that takes care of the Posix based communication with the environment and is the same for all of the monitors targeted in this paper.
- A verified state transition generated out of Coq and all depending definitions.
- A file that realizes glue code for the interaction between the generic part and the generated state transition function. Naturally this file has to be adapted for each individual monitor. It contains abstractions (cf. Section 4.1), e.g., of network packets to events. The abstractions can be verified optionally.

6 Certified Monitors with Coq

We describe how we obtain certified monitors using Coq: the formalization of a monitor, of regular expressions, monitor extraction and the correctness proofs.

6.1 Formalization of Monitor Correctness in Coq

We have adapted an existing library (definitions, morphisms and lemma) for handling regular expressions in Coq in order to work with our definitions of events [2]. Small adaptations were necessary since the original library only handles regular expressions of a particular String type.

We define the following artifacts in Coq:

- Possible events as abstract data types.
- States of monitors as data types and state transition functions written in a functional style.
- The regular expression that specifies the correctness property.
- The statement that the monitor corresponds to the regular expression, using simulation.

[2] Available at http://coq.inria.fr/pylons/contribs/view/RegExp/v8.4 by Takashi Miyamoto.

OCaml program extraction. Due to the functional nature of our monitors, they can be extracted using the Coq command **Recursive Extraction**. The resulting extraction provides a state-transition function and definitions of the used datatypes and possible auxiliary functions. Extraction follows the Coq definitions. This means that our choice of datatypes can influence the performance of monitors. If non-performant definitions are used (e.g., a definition of natural numbers as abstract datatype using 0 and successor constructors) non-performant implementations will be generated.

6.2 Verification of a Monitor with Respect to a Regular Expression

We describe the verification of a monitor with respect to a regular expression in Coq. We establish monitor correctness and assume a state transition function of the internal state of the monitor (cf. Section 4):

$$m_{Step} : (m_{States} \times \Sigma_a) \rightarrow (m_{States} \times bool).$$

To verify monitor correctness for a property ϕ, we first establish a simulation relation R between states (m_{States}) and regular expressions of type Φ – each alphabet of events instantiates a parameterized type of regular expressions; logically this results in an distinguishable types for each alphabet – with $\phi \in \Phi$:

$$R : m_{States} \times \Phi \rightarrow bool$$

The relation performs a check on the semantic correspondence of states using the $\simeq$ relation on regular expressions (see Section 4.2). Finding regular expressions corresponding to a state is done using Arden's Lemma [Ard61]. The proof is done using an induction. The initial case is resolved by using (1). The induction step is proven using the property (2) described below. The following items have to be proven:

1. The first property that is proven states that the initial state of m_{Step} and ϕ are in the simulation relation. This is the basis for proving that m accepts the same lists of events as specified by ϕ.
2. We prove the following property (step-relation correspondence) using the implication $\longrightarrow$:

$$\begin{array}{l} \forall\ m\ m' : m_{States} \quad \phi\ \phi' : \Phi \quad e : \Sigma_a \quad b : bool\ . \\ m_{Step}(m, e) = (m', b) \longrightarrow \phi' = \phi/e \longrightarrow R(m, \phi) \longrightarrow \\ R(m', \phi') \end{array}$$

   It states: for a regular expression ϕ and a state m in the simulation relation R, the succeeding state m' after processing one event e and a succeeding regular expression ϕ' are in the simulation relation again. ϕ' corresponds to accepting (using the / operator) the same event e on ϕ. A Coq formalization of this property is shown in Fig. 5. We use `s`, `s'` to indicate states of type `StatesAutomaton1`. The message alphabet has type `MAa1`. The `=R=` is a relation denoting equivalent regular expressions. `derive` realizes the / operator.

```
Lemma step_correspondence :
 forall (e : MAa1) (s s' : StatesAutomaton1) (r r' : RegExp MAa1),
   s' = (fst(A1Step s e)) ->
   r' =R= derive MAa1 dec_MAa1 e r ->
   regexp_states_rel_a1 s r ->
   regexp_states_rel_a1  s' r'.
```

Fig. 5. Coq formalization of step-relation correspondence

Our correctness criterion for safety invariants states that each finite prefix of event streams (our regular expressions typically work on potentially infinite streams) will only result in non-error states iff the regular expression gets non empty. We prove a stronger property first which implies the correctness criterion and requires that all encountered states and the acceptance state of the regular expressions are in the relation.

Non-safety properties are characterized by the fact that finite prefixes of a trace do not have to fulfill the property even if the entire trace does. It remains possible to prove an adequate simulation relation, and, based on this derive that at least in distinct states a property holds using the method described above.

Example relation. An example relation for the property checked by the automaton from Fig. 3 associates states to their corresponding regular expressions:

$$\begin{aligned} \text{S1} &\simeq (S_1 + T.S_1)^*.(T + \epsilon) \\ \text{S2} &\simeq (S_1.(S_1 + T.S_1)^*.(T + \epsilon)) + \epsilon \\ \text{ERROR} &\simeq \mathit{Empty} \end{aligned}$$

Figure 6 shows the same simulation relation in Coq. The events and states have slightly different names to make them usable together with other automata in the same file. In addition, we also have defined some abbreviations to make the look of constructors Star, Atom, Eps closer to the mathematical notations during the interactive proofs. It can be seen that regular expressions used inside the relation can become larger than the original property. In case of wrong relations, however, we will not be able to establish an overall correctness proof. Thus, the size of the relation does not enlarge the trusted computing base of our approach.

6.3 Verification of Abstractions

It is convenient to deliver only certain events to a monitor. For this reason, we have introduced abstraction functions in Section 4. We verify abstractions by specifying them in Coq and proving the required properties, e.g., for an abstraction function *abs* and an abstract event e_a and a set of possible concrete events Σ_c:

$$\forall\ e_c\ :\ \Sigma_c\ .\ \ e_c = \mathit{specification\ of\ concrete\ events} \longrightarrow e_a = \mathit{abs}(e_c)$$

Proofs are straightforward. The extraction of OCaml code from Coq has to take into account that datatypes are sometimes defined in a different way in Coq and OCaml. For instance, integers are defined using native processor arithmetics in OCaml, but are realized as an inductive datatype in Coq.

```
Definition regexp_states_rel_a1
   (s : StatesAutomaton1) (r : RegExp MAa1) : Prop :=
match s , r with
| A1S1 , x => ((Star MAa1
      ((Atom MAa1 S_a1) || ((Atom MAa1 T_a1)  ++ (Atom MAa1 S_a1) ) ))
    ++ ((Atom MAa1 T_a1) || Eps MAa1)
    ) =R= x
| A1S2 , x => (((Atom MAa1 S_a1 ++
    Star MAa1
      (Atom MAa1 S_a1
       || (Atom MAa1 T_a1 ++ Atom MAa1 S_a1)))
   ++ (Atom MAa1 T_a1 || Eps MAa1)) || Eps MAa1)=R= x
| A1SError , x => Empty MAa1 =R= x
end.
```

Fig. 6. Simulation relation in Coq

7 Evaluation

We evaluate our approach with respect to three criteria that are used in a first step aiming to assess the feasibility of certified RV in an industrial context: proving effort, integration into a bus simulator and performance of monitors.

Proving effort for regular expression based monitors. For proving a new regular expression based monitor correct, one has to establish a relation comprising states and corresponding regular expressions. The main effort is the proof of step-relation correspondence which requires a case distinction on possible states and events ($|states| \times |events|$ different cases). Each case essentially requires some rewriting of equivalent regular expressions to prove the correspondence. This can require several lines of proof code for each event. Automation using tactics might be possible, but require some clever rewriting strategies which we have not developed currently. The other parts of the proofs are either relatively easy or can be reused (are generic) with some adaptations.

Generation of monitors from Coq specifications. We have generated monitor state-transition functions out of our verified Coq formalizations. Coq allows the extraction of executable state-transition functions. Extraction works recursively, so all required types and depending functions are also extracted from their Coq specifications.

Integration of monitors into a bus simulator. We have demonstrated the applicability of our approach by an implementation of a bus simulator for the Rain-Sensor scenario from Section 3.1. The Rain-Sensor senses the rain intensity, sends the intensity value to a Wiper-Controller, which analyses the value and sends control values to a Wiper-Actuator. The example application is implemented in C++ using BSD Sockets for communication over UDP/IP. Only the Wiper-Controller is monitored. The communication between the controller and its monitor is based on Unix pipes. This solution can be used with any Posix

compatible protocol. For example ethernet based bus implementations that fulfill real-time constraints (e.g., [SKS10]).

Performance evaluation of OCaml based monitors. We have built an experimental setup to evaluate the performance of certified monitoring code. A trace generator creates multiple traces and send them to a certified monitor for analysis. Since the performance of the embedded hardware we are aiming at is still subject to change, we have conducted the experiments on different standard machines: Machine 1 is a Macbook Pro i7 at 2GHz, Machine 2 is a Pentium D at 3GHz, Machine 3 is Machine 2 down-clocked at 850MHz. Results are given in Table 1. In each cell, the indicated result (in seconds) is obtained by taking the mean value after a hundred executions. The table shows five properties. The first column shows the property under consideration. The entry **|tr.|** denotes the length of the traces sent to the monitor. The entry **no mon** (resp. **mon**) denotes the execution time in seconds when the trace is not monitored (resp. is monitored) by the certified monitors. The entry **ovhd** indicates the overhead induced by the monitor on the original system: $\frac{\textbf{mon - no mon}}{\textbf{no mon}}$. The entry **kevt/s** indicates the throughput of the monitor, i.e., how many thousands of events it can handle in a second. The timing properties are taken from Section 3.2. The monitors ϕ_4 and ϕ_5 monitor the rain-intensity as explained in Section 3.1.

Timings in Table 1 clearly substantiate our claim that the performance of certified runtime monitors is good. The overhead induced by the monitoring code on the initial system is negligible. This is due to the performance of the optimized code generated by the OCaml compiler. The throughput of the monitors is also

Table 1. Performance evaluation of certified monitors

ϕ	|tr.|	Machine 1				Machine 2				Machine 3			
		no mon	mon	ovhd	kevt/s	no mon	mon	ovhd	kevt/s	no mon	mon	ovhd	kevt/s
ϕ_1	10^4	.7055	.7056	.00022	438.94	.7119	.7119	0	75.98	2.3357	2.3357	0	81.69
	10^5	1.9884	1.9920	.0019		5.6453	5.6453	0		18.585	18.585	0	
	10^6	14.599	14.599	0		55.061	54.451	.0027					
ϕ_2	10^4	.7047	.7047	0	449.76	.7093	.7138	0	73.28	2.1986	2.2474	.03077	75.52
	10^5	2.0095	2.0095	0		5.6589	5.6469	.0027		17.5597	17.898	.0318	
	10^6	14.387	14.436	.003588		54.48	54.690	0.00405					
ϕ_3	10^4	.7050	.7051	.00022	445.79	.712	.7167	.00816	77.88	2.2309	2.2882	.03402	74.79
	10^5	2.0981	2.1032	.00267		5.6476	5.6738	.00539		17.465	18.207	.057	
	10^6	14.507	14.517	.00125		54.48	54.789	0					
ϕ_4	10^4	.7054	.7054	0	441.73	.7085	.7136	.00895	77.16	2.234	2.3109	.04426	73.85
	10^5	2.1334	2.1338	.00036		5.6778	5.6778	0		17.524	18.077	.041	
	10^6	14.343	14.343	0		54.974	54.974	0					
ϕ_5	10^4	.7047	.7051	.0064	450.32	.7093	.7127	.00652	74.07	2.1978	2.2261	.02127	78.92
	10^5	2.0754	2.0762	.00057		5.6273	5.6273	0		17.724	17.827	.01723	
	10^6	14.502	14.502	0		54.729	54.809	0.00204					

very satisfactory. Actually, the similar performance results observed on Machine 2 and Machine 3 made us notice that the throughput of the monitor is actually not limited by the monitoring code but by the performance of the OS primitives used to establish communication between the system and the monitors. Note that, for traces of length 10^6, some erratic measures were taken on Machine 3, probably because of down-clocking. Thus, we could not report reliable numbers.

8 Conclusion and Future Work

We presented work towards certified RV by means of higher-order theorem provers. In particular, we have demonstrated the feasibility to generate OCaml based runtime monitors out of verified Coq formalizations. We have demonstrated the deployment in a bus simulator. Furthermore, we have presented a performance evaluation of OCaml based monitors in general. Our properties can be checked in an acceptable time and it seems feasible to deploy the demonstrated solutions in industrial domains. The shown aspects are an important prerequisite for demonstrating the feasibility for large scale applications.

Despite being sufficient for the sketched property monitoring of bus messages, the properties regarded in this paper are relatively small and simple. As an academic goal future work should extend the expressiveness and also regard more complex properties. More particularly, it would be of great interest to be able to extract monitoring code for parametric properties, i.e., properties featuring parameterized events taking values at runtime. Recent advances [RC12] in runtime verification that give a semantics to these properties will certainly help. Furthermore, as a goal with a strong engineering focus we want to deploy monitors on real embedded hardware and industrial demonstrators as a next step. Dynamic aspects like adding or removing components during runtime and the impact on monitors are other aspects.

References

[AAC+05] Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to AspectJ. SIGPLAN Notices (October 2005)

[Ard61] Arden, D.N.: Delayed-logic and finite-state machines, pp. 133–151. IEEE (1961)

[BGHS04] Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)

[BGHS10] Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. Journal of Aerospace Computing, Information, and Communication (2010)

[BH11] Barringer, H., Havelund, K.: TraceContract: A Scala DSL for Trace Analysis. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 57–72. Springer, Heidelberg (2011)

[BRH10] Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: from EAGLE to RuleR. Journal of Logic and Computation (June 2010)

[BG11] Blech, J.O., Grégoire, B.: Certifying compilers using higher-order theorem provers as certificate checkers. Formal Methods in System Design (2011)

[CPS09] Colombo, C., Pace, G.J., Schneider, G.: Larva — safer monitoring of real-time Java programs (tool paper). In: Software Engineering and Formal Methods (SEFM 2009), pp. 33–37. IEEE Computer Society (2009)

[Coq12] The Coq development team. The coq proof assistant reference manual v8.4 (2012), http://coq.inria.fr

[FFM09] Falcone, Y., Fernandez, J.-C., Mounier, L.: Runtime Verification of Safety-Progress Properties. In: Bensalem, S., Peled, D.A. (eds.) RV 2009. LNCS, vol. 5779, pp. 40–59. Springer, Heidelberg (2009)

[FRay05] FlexRay communications system protocol specification version 2.1, FlexRay Consortium (May 2005)

[HG05] Havelund, K., Goldberg, A.: Verify Your Runs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 374–383. Springer, Heidelberg (2008)

[KW] Kaiser, R., Wagner, S.: The PikeOS Concept: History and Design. SysGO AG White Paper, http://www.sysgo.com

[KVL+99] Kim, M., Viswanathan, M., Lee, I., Ben-Abdellah, H., Kannan, S., Sokolsky, O.: Formally Specified Monitoring of Temporal Properties. In: European Conference on Real-Time Systems (ECRTS) (1999)

[Kop93] Kopetz, H.: TTP - A time-triggered protocol for fault-tolerant real-time systems. In: Fault-Tolerant Computing. IEEE (1993)

[LKK+99] Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime Assurance Based On Formal Specifications. In: International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas (1999)

[MJG+11] Meredith, P., Jin, D., Griffith, D., Chen, F., Roşu, G.: An overview of the MOP runtime verification framework. International Journal on Software Tools for Technology Transfer (2011)

[PZ06] Pnueli, A., Zaks, A.: PSL Model Checking and Run-Time Verification Via Testers. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 573–586. Springer, Heidelberg (2006)

[RC12] Rosu, G., Chen, F.: Semantics and Algorithms for Parametric Monitoring. Logical Methods in Computer Science (2012)

[Rus08] Rushby, J.: Runtime Certification. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 21–35. Springer, Heidelberg (2008)

[SKS10] Steinbach, T., Korf, F., Schmidt, T.C.: Comparing time-triggered Ethernet with FlexRay: An evaluation of competing approaches to real-time for invehicle networks. In: 8th IEEE International Workshop on Factory Communication Systems (2010)

[SB06] Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: Proc. of the 5th Int. Workshop on Runtime Verification, RV 2005. ENTCS, vol. 144(4). Elsevier (2006)

[SH11] Sridhar, M., Hamlen, K.W.: Flexible in-lined reference monitor certification: challenges and future directions. In: Programming Languages meets Program Verification. ACM (2011)

Author Index