Shaoying Liu
Tom Maibaum
Keijiro Araki (Eds.)

LNCS 5256

Formal Methods and Software Engineering

10th International Conference
on Formal Engineering Methods, ICFEM 2008
Kitakyushu-City, Japan, October 2008, Proceedings

Springer

Lecture Notes in Computer Science 5256

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

Editorial Board

Shaoying Liu Tom Maibaum
Keijiro Araki (Eds.)

Formal Methods and Software Engineering

10th International Conference
on Formal Engineering Methods, ICFEM 2008
Kitakyushu-City, Japan, October 27-31, 2008
Proceedings

Springer

Volume Editors

Shaoying Liu
Hosei University, Faculty of Computer and Information Sciences
3-7-2 Kajino-cho Koganei-shi, Tokoy 184-8584, Japan
E-mail: sliu@hosei.ac.jp

Tom Maibaum
McMaster University, Department of Computing and Software
1280 Main St West, Hamilton ON, L8S 4K1, Canada
E-mail: tom@maibaum.org

Keijiro Araki
Kyushu University
Department of Computer Science and Communication Engineering
Graduate School of Information Science and Electrical Engineering
744 Motooka, Nishi-ku, Fukuoka 812-8581,Japan
E-mail: araki@csce.kyushu-u.ac.jp

Library of Congress Control Number: 2008937804

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

LNCS Sublibrary: SL 6 – Image Processing, Computer Vision, Pattern Recognition, and Graphics

ISSN	0302-9743
ISBN-10	3-540-88193-X Springer Berlin Heidelberg New York
ISBN-13	978-3-540-88193-3 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12538577 06/3180 5 4 3 2 1 0

Preface

Formal engineering methods are intended to offer effective means for integration of formal methods and practical software development technologies in the context of software engineering. Their purpose is to provide effective, rigorous, and systematic techniques for significant improvement of software productivity, quality, and tool supportability. In comparison with formal methods, a distinct feature of formal engineering methods is that they emphasize the importance of the balance between the qualities of simplicity, visualization, and preciseness for practicality. To achieve this goal, formal engineering methods must be developed on the basis of both formal methods and existing software technologies in software engineering, and they must serve the improvement of the software-engineering process.

ICFEM 2008 marks the tenth anniversary of the first ICFEM conference, which was held in Hiroshima in 1997. It aims to bring together researchers and practitioners who are interested in the development and application of formal engineering methods to present their latest work and discuss future research directions. The conference offers a great opportunity for researchers in both formal methods and software engineering to exchange their ideas, experience, expectation and to find out whether and how their research results can help advance the state of the art.

This volume contains the papers presented at ICFEM 2008, held October 27–31, 2008 at the Kitakyushu International Conference Center, Kitakyushu City, Japan. There were 62 submissions, each of which was reviewed by three Program Committee members. The committee decided to accept 20 papers based on originality, technical contribution, presentation, and relevance to formal engineering methods. We sincerely thank the Program Committee members and their co-reviewers for their professional work and great effort in the paper reviewing and selection process. We also thank the three keynote speakers, Takuya Katayama, Jeff Offutt, and John Hatcliff, for their contributions to the conference program. Professor Katayama gave a talk on how formal methods can be made acceptable by industry. Professor Offutt presented a talk on how programmers and testers could use formal methods in practice; and Professor Hatcliff spoke about contract-based reasoning for verification and certification of secure information-flow policies in industrial products.

In addition to the conference's main program, two workshops were organized. One was the First IEEE International Workshop on UML and Formal Methods (UML&FM 2008) and the other was the First International Workshop on Formal Methods Education and Training (FMET 2008). We thank the workshop organizers for their great efforts and contributions to the conference.

ICFEM 2008 was jointly organized by Kyushu University and Hosei University. It was sponsored by the ICFEM Organizing Committee and supported by

the IEEE Fukuoka Section, the Software Engineers Association of Japan (SEA), the West Japan Industry and Trade Convention Association and several other organizations. The EasyChair system was used to manage the submissions, reviewing, paper selection, and proceedings production. We would like to thank the EasyChair team for a very useful tool.

August 2008

Shaoying Liu
Tom Maibaum
Keijiro Araki

Organization

Conference Chairs

General Chair	Keijiro Araki	Kyushu University, Japan
Program Chairs	Shaoying Liu	Hosei University, Japan
	Tom Maibaum	McMaster University, Canada
Publicity Chair	Yoichi Omori	Kyushu University, Japan
Tutorial Chair	Fumiko Nagoya	Hosei University, Japan

Program Committee

Nazareno Aguirre	Colin Fidge	Anders Ravn
Toshiaki Aoki	John Fitzgerald	Ken Robinson
Keijiro Araki	Marcelo Frias	Shin Sahara
David Basin	Stefania Gnesi	Davide Sangiorgi
Michael Butler	Mike Hinchey	Wuwei Shen
Ana Cavalcanti	Soon-Kyeong Kim	Jing Sun
Steve Cha	Peter Gorm Larsen	Koichi Takahashi
Jessica Chen	Kung-Kiu Lau	Testuo Tamai
Yuting Chen	Mark Lawford	T.H. Tse
Yoonsik Cheon	Michael Leuschel	Farn Wang
S.C. Cheung	Xuandong Li	Jim Woodcock
Peter J. Clarke	Zhiming Liu	Wang Yi
Jim Davies	Huaikou Miao	Jian Zhang
Jin Song Dong	Shin Nakajima	Hong Zhu
Zhenhua Duan	Michael Poppleton	

Local Organization

Keijiro Araki
Masumi Toyoshima
Shigeru Kusakabe

External Reviewers

Jean-Raymond Abrial	Gihwon Kwon	Daniel Plagge
Andrew Allen	Woo-Jin Lee	Shengchao Qin
Djuradj Babich	Florian Letombe	GermÃn Regis
Richard Banach	Hui Liang	Yuxiang Shi
Heung-Seok Chae	Yang Liu	Jane Sinclair

Zhenbang Chen	Paulo Matos	Colin Snook
Wei Chen	Franco Mazzantti	Christoph Sprenger
Alessandro Fantechi	Manuel Mazzara	Jun Sun
Bernd Fischer	Hiroshi Mochio	Kenji Taguchi
Dilian Gurov	Charles Morisset	Toshinori Takai
Thai Son Hoang	Tsz-Hin Ng	Izumi Takeuti
Christian Haack	Naoya Nitta	James Welch
Eun Young Kang	Ioannis Ntalamagkas	Letu Yang
Raman Kazhamlakin	Joseph Okika	Kenro Yatake
John Knudsen	Yoichi Omori	Jianhua Zhao
Pavel Krcal	Lucian Patcas	Xian Zhang
Shigeru Kusakabe	Mark Pavlidis	

ICFEM Steering Committee

Keijiro Araki	Kyushu University, Japan
Jin Song Dong	National University, Singapore
Chris George	UNU-IIST, Macao
Jifeng He	Chair, East China Normal University, China
Mike Hinchey	University of Limerick, Ireland
Shaoying Liu	Hosei University, Japan
John McDermid	University of York, UK
Tetsuo Tamai	University of Tokyo, Japan
Jim Woodcock	University of York, UK

Table of Contents

Invited Talks

Specification and Verification

Testing

Verification 1

Model Checking and Analysis

Verification 2

Tools

Application of Formal Methods

Semantics

How Can We Make Industry Adopt Formal Methods?

Takuya Katayama

Japan Advanced Institute of Science and Technologies

Despite a long history of technical development of formal methods and their success in advanced system developments, they are not well recognized nor accepted as effective and standard methodology in industrial and commercial software development. This fact should be considered seriously if we evaluate their intrinsic technical superiority. Though ignorance of the new technologies will be one reason for the reluctance of industry to use formal methods, we should also check if current formal methods will be enough for the real software development practices in industry. In this talk, based on interviews with software engineers and managers, these issues will be addressed including proposal of formal methods adoptable in industry.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, p. 1, 2008.

Programmers Ain't Mathematicians, and Neither Are Testers

Jeff Offutt

George Mason University, USA

Formal methods have been developed for decades. An early promise was that we could use formal methods to "prove" our programs correct. We have also tried to use formal methods to completely specify functional behavior of programs and to partially specify specific aspects of software behavior. Research into formal methods have led to weaker techniques to "model" functional behavior, less completely and less precisely, but in ways that are easier to use. Despite these years of activity, formal methods are still seldom used in industry. As software engineering researchers, we are compelled to take the view that there must be a path from our research to actual use in industry, where real software developers use our ideas to help create real, and better, software. Thus we must ask, are formal methods a solution in search of a problem?

In this talk, I will draw a distinction between mathematical thinking, which is required to develop formal models of software behavior, and engineering thinking, which is required to develop working software. In a broad sense, formal methods are used to create abstractions, and abstraction should be used to handle complexity, not to ignore it. The talk will explore this distinction and suggest specific ways formal methods can be successfully integrated into software development, testing and education. Mathematicians who specify software models and design software tests should delve into this abstraction, using formal engineering methods to help real programmers build real software, faster, better and cheaper.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, p. 2, 2008.

Contract-Based Reasoning for Verification and Certification of Secure Information Flow Policies in Industrial Workflows*

John Hatcliff

SAnToS Laboratory
Kansas State University
Manhattan, KS 66506, USA
hatcliff@cis.ksu.edu
http://www.cis.ksu.edu/santos

Abstract. Successful transfer of formal engineering methods from academia to industrial development depends on a variety of factors: a proper understanding of the industrial development context, effective and usable technology that can be integrated with development workflows to provide a compelling solution to serious development challenges, "buy-in" from industrial developers and management, an appropriate business model for supporting the deployed technology, plus a lot of luck. I describe how many of these factors are manifesting themselves in an effort by our research group to transition rigorous static analyses and novel Hoare-style logics into a large industrial development process for information assurance and security applications.

The applications that we are targeting address the following problem: international infrastructure and defense forces are increasingly relying on complex systems that share information with multiple levels of security (MLS). In such systems, there is a strong tension between providing aggressive information flow to gain operational and strategic advantage while preventing leakage to unauthorized parties. In this context, it is exceedingly difficult to specify and certify security policies, and produce *evidence* that a system provides end-to-end trust.

In the past, verification and certification obligations in this domain have been met by using heavy-weight theorem proving technology that requires many manual steps or by light-weight contract-based static analyses that are too imprecise for specifying and verifying crucial information flow properties. In this talk, I will explain how our research team is (a) building integrated tool support for automatically discovering and visualizing information flows through programs and architectures, and (b) providing code-integrated software contracts for specifying information flow policies, and (c) applying synergistic blends of static analyses and automated reasoning based on weakest-precondition calculi to aid developers in automatically discharging verification obligations. These techniques aim to hit a "sweet spot" that provides greater automation and developer integration

* This work was supported in part by the US National Science Foundation (NSF) awards 0454348 and CAREER award 0644288, the US Air Force Office of Scientific Research (AFOSR), and Rockwell Collins.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 3–4, 2008.

than previous theorem-proving-based approaches while offering increased precision over previous static-analysis-based frameworks. Throughout the presentation, I will assess approaches/strategies that have been successful in moving our research results into industrial practice and summarize challenges that remain.

Acknowledgments

This talk is based on joint work with researchers from Kansas State including Torben Amtoft, Robby, Edwin Rodríguez, Jonathan Hoag, Todd Wallentine and Loai Zomlot, and with David Greve from Rockwell Collins, Advanced Technology Center.

References

1. Amtoft, T., Bandhakavi, S., Banerjee, A.: A logic for information flow in object-oriented programs. In: 33rd Principles of Programming Languages (POPL), pp. 91–102 (2006)
2. Amtoft, T., Banerjee, A.: A logic for information flow analysis with an application to forward slicing of simple imperative programs. Science of Comp. Prog. 64(1), 3–28 (2007)
3. Amtoft, T., Banerjee, A.: Verification condition generation for conditional information flow. In: 5th ACM Workshop on Formal Methods in Security Engineering (FMSE), pp. 2–11 (2007); A long version, with proofs, appears as technical report KSU CIS TR 2007-2
4. Amtoft, T., Hatcliff, J., Rodríguez, E., Robby,, Hoag, J., Greve, D.: Specification and checking of software contracts for conditional information flow. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014. Springer, Heidelberg (2008)
5. Chapman, R., Hilton, A.: Enforcing security and safety models with an information flow analysis tool. In: SIGAda 2004, Atlanta, Georgia, pp. 39–46. ACM Press, New York (2004)
6. Greve, D., Wilding, M., Vanfleet, W.M.: A separation kernel formal security policy. In: 4th International Workshop on the ACL2 Prover and its Applications (ACL2 2003) (2003)
7. Heitmeyer, C.L., Archer, M., Leonard, E.I., McLean, J.: Formal specification and verification of data separation in a separation kernel for an embedded system. In: 13th ACM Conference on Computer and Communications Security (CCS 2006), pp. 346–355 (2006)
8. Ranganath, V.P., Amtoft, T., Banerjee, A., Hatcliff, J., Dwyer, M.B.: A new foundation for control dependence and slicing for modern program structures. TOPLAS 29(5) (August 2007); In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444. Springer, Heidelberg (2005)
9. Ranganath, V.P., Hatcliff, J.: Slicing concurrent Java programs using Indus and Kaveri. International Journal on Software Tools for Technology Transfer (STTT) 9(5-6), 489–504 (2007); Special section FASE 2004/2005.
10. Rushby, J.: The design and verification of secure systems. In: 8th ACM Symposium on Operating Systems Principles, vol. 15(5), pp. 12–21 (1981)
11. Rushby, J., DeLong, R.: Compositional security evaluation: The MILS approach, `http://www.csl.sri.com/~rushby/slides/iccc07.pdf`

Specifying and Verifying Event-Based Fairness Enhanced Systems

Jun Sun[1], Yang Liu[1], Jin Song Dong[1], and Hai H. Wang[2]

[1] School of Computing,
National University of Singapore
{sunj,liuyang,dongjs}@comp.nus.edu.sg
[2] School of Electronics and Computer Science,
University of Southampton
hw@ecs.soton.ac.uk

Abstract. Liveness/Fairness plays an important role in software specification, verification and development. Existing event-based compositional models are safety-centric. In this paper, we describe a framework for systematically specifying and verifying event-based systems under fairness assumptions. We introduce different event annotations to associate fairness constraints with individual events. Fairness annotated events can be used to embed liveness/fairness assumptions in event-based models flexibly and naturally. We show that state-of-the-art verification algorithms can be extended to verify models under fairness assumptions, with little computational overhead. We further improve the algorithm by other model checking techniques like partial order reduction. A toolset named PAT has been developed to verify fairness enhanced event-based systems. Experiments show that PAT handles large systems with multiple fairness assumptions.

1 Introduction

Critical system requirements like safety, liveness and fairness play important roles in system/software specification, verification and development. Safety properties ensure that something undesirable never happens. Liveness properties state that something desirable must eventually happen. Fairness properties state that if something is enabled sufficiently often, then it must eventually happen. Often, fairness assumptions are necessary to prove liveness properties.

Over the last decades, specification and verification of safety properties have been studied extensively. There have been many languages and notations dedicated to safety-critical systems, e.g., Z, VDM, CCS and CSP. The concept of liveness itself is problematic [17]. Fairness constraints have been proved to be an effective way of expressing liveness, not mentioning that itself is important in system specification and verification. For instance, without fairness constraints, verifying of liveness properties may often produce counterexamples which are due to un-fair executions, e.g., a process or choice is infinitely ignored. State-based fairness constraints have been well studied in automata theory based on accepting states, e.g., in the setting of Büchi/ Rabin/Streett/Muller automata. It has been observed that the notion of fairness is not easily combined with the bottom-up type of compositionality (of process algebra for instance [23]), which is important for attacking the complexity of system development.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 5–24, 2008.

A common practice of verifying liveness relies on explicitly stating all fairness assumptions as premises of the liveness properties. This approach is not feasible if there are many fairness constraints. Note that it is relatively straightforward to verify whether the system satisfies a fairness property. It is verification under multiple fairness assumptions which may be infeasible. For instance, a method to prove that a program satisfies some property is Linear Temporal Logic (LTL) model checking. Given an LTL formula ϕ, the model checker transforms the negation of ϕ into a Büchi automaton, builds the product of the automaton and the program and then checks this product for emptiness. The size of the constructed Büchi automaton is exponential to the length of ϕ. A formula composing of many fairness premises results in a huge Büchi automaton and thus makes model checking infeasible. For example, SPIN is a rather popular LTL model checker [15]. The algorithm it uses for generating Büchi automata handles only a limited number of fairness constraints. The following table shows experiments on the time and space needed for SPIN to generate the automaton from standard notion of fairness, in particular, justice and compassion [16].

Prop.	n	Time (Sec.)	Memory	#Büchi States
$(\bigwedge_{i=1}^{n} \Box\Diamond p_i) \Rightarrow \Box\Diamond q$	1	0.08	466Kb	74
same above	3	4.44	27MB	1052
same above	5	more than 3600	more than 1Gb	–
$(\bigwedge_{i=1}^{n}(\Box\Diamond p_i \Rightarrow \Box\Diamond q_i)) \Rightarrow \Box\Diamond s$	1	0.13	487.268	134
same above	2	1.58	10123.484	1238
same above	3	30.04	55521.708	4850
same above	4	4689.24	more than 1Gb	–

The experiments are made on a 3.0GHz Pentium IV CPU and 1 GB memory executing SPIN 4.3. The results show that it takes a non-trivial amount of time to handle 5 fairness constraints. In order to overcome this problem, SPIN offers an option to handle weak fairness on the level of processes. However, it may not be always sufficient. Process-level fairness states that all enabled events from different processes must be engaged or disabled eventually, which is overwhelming sometimes. For instance, it is reasonable to require that a submitted request must eventually be served, while it is not to require that always eventually there is a request. Another approach [16] to model check under fairness assumptions is to model fairness using global accepting states (or in the form of justice/compassion condition [16]). For instance, a run accepted by a Büchi automaton must visit at least one accepting state infinitely often. This approach is not applicable to event-based compositional systems.

In [17], a language independent definition of event-based fairness has been proposed and studied. Let e be an event. Let $\Diamond$, $\Box$ be the temporal operators which informally reads as 'eventually' and 'always' respectively.

$$wf(e) \mathrel{\widehat{=}} \Diamond\Box(e \ is \ enabled) \Rightarrow \Box\Diamond(e \ is \ engaged)$$
$$sf(e) \mathrel{\widehat{=}} \Box\Diamond(e \ is \ enabled) \Rightarrow \Box\Diamond(e \ is \ engaged)$$

The weak fairness $wf(e)$ asserts that if an event e eventually becomes enabled forever, infinitely many occurrences of the event steps must be observed. The strong fairness

$sf(e)$ asserts that if e is infinitely often enabled (or in other words, repeatedly enabled), infinitely many occurrences of the event must be observed. Strong fairness implies weak fairness. A system satisfies a fairness constraint if and only if every run of the system satisfies the fairness constraint.

Partly inspired by the work above, we propose an alternative approach for specifying fairness constraints, with efficient verification in mind. Instead of stating the fairness assumptions as a part of the property to verify or additional global accepting states, they are embedded in the compositional models. We introduce different event annotations to associate fairness assumption with individual events. Event-based annotation allows us to model fairness naturally and flexibly. For instance, by annotating all events weak fair, we achieve process-level fairness (as offered in SPIN). Next, we investigate automated verification of models with event-based fairness. We show that existing state-of-the-art verification algorithms can be extended to handle event-based fairness with little computational overhead. An on-the-fly model checking algorithm is developed. The algorithm is further improved by techniques like partial order reduction. A toolset named PAT (stands for Process Analysis Toolset) has been developed to realize the algorithms (which also supports functionalities like standard model checking, model simulation, etc). Experiment results show that PAT handles non-trivial systems with multiple fair constraints efficiently.

Our contribution includes an approach to model fairness in event-based compositional system models and an on-the-fly model checking algorithm with partial order reduction for verifying those models. This paper is related to works on specification and verification of liveness and fairness, e.g., verification under weak fairness in SPIN and early works discussing liveness associated with events in the framework of Promela, CCS or CSP, evidenced in [15, 7, 6, 21]. One way of capturing fairness is to alter the language semantics so that all events are fairly treated (e.g., [15, 21, 3]), i.e., the semantics of parallel composition is enhanced to be fair. From our point of view, the difference between *parallel* and *sequential* processes may be irrelevant and fairness shall be independent of one particular operator. In practice, it may be that only certain events need to fulfill fairness constraints. For instance, a common requirement is that "some action must eventually occur if some other action occurs" (i.e., compassion conditions). In other works [7, 6], events or processes are annotated with special markings to capture fairness. However, previous approaches do not easily combine with the bottom-up compositionality of process algebra, i.e., the fairness constraints may be lost once the process is composed with others. In this work, annotations are used to associate different kinds of fairness assumptions with relevant events in the relevant module/process, which may yet has global effects. Moreover, we develop automated verification support for our notion of fairness. We remark that the concept of event-based fairness is not limited to process algebras. Specification of fairness in programming languages has been discussed in the line of works by Apt, Francez and Katz [2]. Event-based fairness may allow fairness constraints that may not be feasible (and thus violates one of the principles in [2]), which makes our verification techniques crucial. Our works on verification of models with embedded fairness constraints are related to previous works on verification under fairness assumptions [16, 18, 13], in which liveness/fairness constraints are either specified using temporal logic formulae or captured using *global* accepting states.

We use a different way of specifying fairness constraints and hence our model checking algorithm is different from theirs. We also extend our algorithm with techniques like partial order reduction to handle large systems. This work is remotely related to our previous works on verification of event-based specifications [10, 9, 19].

The remainder of the paper is organized as follows. Section 2 reviews the input language of PAT and its semantics. Section 3 introduces our event annotations. Section 4 presents and evaluates the on-the-fly model checking algorithm and partial order reduction. Section 5 concludes the paper.

2 Background

Without loss of generality, we present our ideas using a simple compositional language which supports concurrency, multi-threaded synchronization, shared variables and assignments. In the next section, we will extend this language with fairness annotations.

A model is composed of a set of global variables and a set of process definitions. One of the processes is identified by the starting process (as the main method in Java), which captures the system behaviors after initialization. A process is defined as using the following constructs. Most of the compositional operators are borrowed from the classic CSP [14].

$$P \mathrel{\widehat{=}} \mathit{Stop} \mid \mathit{Skip} \mid e \rightarrow P_1 \mid P_1;\ P_2 \mid P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \\ \mid P_1 \lhd b \rhd P_2 \mid [b] \bullet P \mid P_1 \triangle P_2 \mid P_1 \,|[\, X \,]|\, P_2$$

where b is a Boolean expression, X is a set of events and e is an event. Note that e could be an abstract event (single or compound) or an assignment (e.g., $x := x + 1$). Process Stop does nothing but deadlocks. Process Skip terminates successfully. Event prefixing $e \rightarrow P$ is initially willing to engage in event e and behaves as P afterward. $\mathit{Skip} = \checkmark \rightarrow \mathit{Stop}$ where $\checkmark$ is the termination event. The sequential composition, $P_1;\ P_2$, behaves as P_1 until its termination and then behaves as P_2. A choice between two processes is denoted as $P_1 \,\Box\, P_2$ (for external choice) or $P_1 \sqcap P_2$ (for nondeterminism). A choice depending on the truth value of a Boolean expression is written as $P_1 \lhd b \rhd P_2$. If b is true, the process behaves as P_1, otherwise P_2. The state guard $[b] \bullet P$ is blocked until b is true and then proceeds as P, i.e., a guarded command. $P_1 \triangle P_2$ behaves as P_1 until the first event of P_2 is engaged, then P_1 is interrupted and P_2 takes control.

One reason for using a CSP-based language is to study fairness assumptions in a setting with multi-threaded lock-step synchronization. Let Σ_P be the alphabet of P which excludes τ (internal action) and $\checkmark$. Note that alphabets can be manually set or by default be the set of events constituting the process expression. Parallel composition of processes is written as $P \,|[\, X \,]|\, Q$. Events in X must be synchronized by both processes. If X is empty, P and Q run in parallel independently. X is skipped if it is exactly $\Sigma_P \cap \Sigma_Q$. Multiple processes may run in parallel, written as $P_1 \parallel P_2 \parallel \cdots \parallel P_n$. Shared events must be synchronized by all processes whose alphabet contains the event. Recursion is allowed by process referencing. The semantics of recursion is defined as Tarski's weakest fixed-point. The valuation of the variables is a set of pairs which map a variable to its current value. A system state is a pair (P, V) where P is a process expression and V is the valuation of the global variables.

Example 1. The classic dining philosophers example [14] is used as a running example.

$$\begin{aligned} Phil(i) &= get.i.(i+1)\%N \rightarrow get.i.i \rightarrow eat.i \rightarrow put.i.(i+1)\%N \rightarrow \\ &\quad put.i.i \rightarrow Phil(i) \\ Fork(i) &= get.i.i \rightarrow put.i.i \rightarrow Fork(i) \ \Box \\ &\quad get.(i-1)\%N.i \rightarrow put.(i-1)\%N.i \rightarrow Fork(i) \\ College(N) &= \Big|\Big|_{i=0}^{N-1} (Phil(i) \parallel Fork(i)) \end{aligned}$$

where N is the number of philosophers, $get.i.j$ ($put.i.j$) is the action of the i-th philosopher picking up (putting down) the j-th fork. Process $Phil(i)$ models the behaviors of the i-th philosopher. Process $Fork(i)$ models the behaviors of the i-th fork. Process $College(N)$ is the indexed parallel composition of multiple philosophers and forks. It is known that $College(N)$ deadlocks when each dining philosopher picks up one fork. By asking one of the philosophers to pick up the forks in a different order, the system becomes deadlock-free. □

We focus on the operational semantics in this paper. The operational semantics for CSP presented in [4] has been extended with shared variables (presented in [28]). The sets of behaviors of processes can equally and equivalently be extracted from the operational semantics, thanks to congruence theorems. Fairness properties state that an event which is either repeatedly or continuously enabled must eventually occur. They therefore affect only the infinite, and not finite, traces of a process. We present an infinite trace semantics, inspired by the infinite trace semantics for CSP [25]. Note that finite traces are extended to infinite ones in a standard way, i.e., by attaching infinite number of idling events to the rear. Let Σ^* and Σ^ω be the set of finite and infinite sequences of events respectively.

Definition 1 (Infinite Traces). *Let P be a process and V be a valuation of the data variables. The set of infinite traces is written as $inftr(P, V)$. An infinite trace $\tilde{tr} : \Sigma^\omega$ is in $inftr(P, V)$ if and only if there exists an infinite sequence of $\tilde{P}$ and $\tilde{V}$ such that*

- *$\tilde{P}(0) = P$ and $\tilde{V}(0) = V$,*
- *for all i, $(\tilde{P}(i), \tilde{V}(i)) \stackrel{\tilde{tr}(i)}{\Rightarrow} (\tilde{P}(i+1), \tilde{V}(i+1))$*

where $\Rightarrow$ is the smallest transition relation defined by the operational semantics [28].

An infinite run of a process P with variable valuation V is an alternating infinite sequence of states and events $(\tilde{P}(0), \tilde{V}(0)), \tilde{a}(0), \cdots (\tilde{P}(i), \tilde{V}(i)), \tilde{a}(i), \cdots$ which conforms to the operational semantics. An infinite sequence of events is a trace if and only if there is a run with the exact same sequence of events. A state (P', V') is reachable from (P, V) if and only if there is a finite run from (P, V) to (P', V').

Definition 2 (Enabledness). *Let P be a process. Let V be a valuation of the data variables. $enabled(P, V) = \{e : \Sigma \mid \exists P', V' \bullet (P, V) \stackrel{e}{\Rightarrow} (P', V')\}$.*

Given P and V, an event is enabled if and only if it is in $enabled(P, V)$. It is disabled if it is not enabled. Note that given a parallel composition $P \,|[\, X \,]|\, Q$, an event in X is enabled in the composition if and only if it is enabled in both P and Q.

It is known that CSP (as well as CCS) lacks the notion of liveness or fairness [14, 6]. An event can be enabled *forever* but never be engaged, or an event may be enabled *infinitely often* but never been engaged. In the paper, we assume properties are stated in the form of LTL formulae. Different from stand LTL, we adopt the work presented in [5] so that events may be used to form LTL formulae. A desirable property for process $College(5)$ is $\Box\Diamond eat.0$, i.e., always eventually the 0-th philosopher eats and thus never starves. Note that this property is not true, i.e., $College(5) \not\models \Box\Diamond eat.0$. The following traces may be returned as counterexamples.

$\langle get.0.1, get.1.2, get.2.3, get.3.4, get.4.0\rangle$ – T0
$\langle get.3.4, get.3.3, eat.3, put.3.4, put.3.3\rangle^{\omega}$ – T1
$\langle get.1.2, get.1.1, eat.1, put.1.2, put.1.1\rangle^{\omega}$ – T2

Assume that given a trace tr, tr^{ω} repeats tr infinitely. T0 is a trace which leads to the deadlock situation, in which case all philosophers starve. T1 and T2 are returned because the model lacks of both weak and strong fairness. In T1's scenario, the 3-rd philosopher greedily gets the forks and eats forever. This counterexample is due to lack of weak fairness, i.e., the event $get.0.1$ is always enabled but never engaged. In T2's scenario, the 1-st philosopher greedily gets the forks and eats forever. This counterexample is due to lack of strong fairness, i.e., the event $get.0.1$ is repeatedly enabled (after event $put.1.1$ and disabled after event $get.1.1$) but never engaged.

In order to verify that the system does satisfy the property under the assumption that the system is (strongly) fair and the deadlock situation never occurs, we may verify the following property,

$(\bigwedge_{i=0}^{N-1} \Box\Diamond get.i.(i+1)\%N)$ – C1
$\wedge\ \Box\Diamond put.1.2$ – C2
$\Rightarrow \Box\Diamond eat.0$

where C1 and C2 are stated as premises of the property. C1 states that each philosopher must always eventually get his first fork. C2 states that one of the philosopher (in this case, the 1-st) must eventually put down a fork. It is used to avoid the deadlock situation. Though this property is true, automata-based verification is deficient because of its size.

3 Event Annotations

In this section, we introduce a way of modeling event-based fairness, i.e., by annotating an event with fairness assumptions. Given an event e, four different annotations can be used to associate different fairness assumptions with e. The annotations are summarized in Table 1. In the following, we discuss them one by one.

A *weak* fair event is written as $wf(e)$. Event $wf(e)$ plays the same role as e except that it carries a weak fairness constraint. That is, if a weak fair event is always enabled, it must be eventually engaged. In other words, the system must move beyond a state where there is a weak fair event enabled. Weak fair events allow us to express weak fairness constraints naturally. It can be shown that both weak and strong fairness are expressible using weak fair events (as strong fairness can be transformed to weak fairness by paying

Table 1. Event-based Fairness Annotations

Annotation	Name	Semantics
$wf(e)$	weak fair event	$\Diamond\Box e$ is enabled $\Rightarrow \Box\Diamond e$ is engaged
$sf(e)$	strong fair event	$\Box\Diamond e$ is enabled $\Rightarrow \Box\Diamond e$ is engaged
$wl(e)$	weak live event	$\Diamond\Box e$ is ready $\Rightarrow \Box\Diamond e$ is engaged
$sl(e)$	strong live event	$\Box\Diamond e$ is ready $\Rightarrow \Box\Diamond e$ is engaged

the price of one variable [16]). However, strong fairness constraints may require more than what fair events can offer in a natural way. Therefore, we introduce the notion of *strong* fair events to capture strong fairness elegantly. A *strong* fair event, written as $sf(e)$, must be engaged if it is repeatedly enabled.

Example 2. The following demonstrates how we may achieve process level weak fairness (as the option offered in SPIN).

$$\begin{aligned}
\mathit{fPhil}(i) &= \mathit{wf}(\mathit{get}.i.(i+1)\%N) \rightarrow \mathit{wf}(\mathit{get}.i.i) \rightarrow \mathit{wf}(\mathit{eat}.i) \\
&\quad \rightarrow \mathit{wf}(\mathit{put}.i.(i+1)\%N) \rightarrow \mathit{wf}(\mathit{put}.i.i) \rightarrow \mathit{fPhil}(i) \\
\mathit{fFork}(i) &= \mathit{wf}(\mathit{get}.i.i) \rightarrow \mathit{wf}(\mathit{put}.i.i) \rightarrow \mathit{fFork}(i) \ \Box \\
&\quad \mathit{wf}(\mathit{get}.(i-1)\%N.i) \rightarrow \mathit{wf}(\mathit{put}.(i-1)\%N.i) \rightarrow \mathit{fFork}(i) \\
\mathit{fCollege}(N) &= \Big\|_{i=1}^{N-1} (\mathit{fPhil}(i) \parallel \mathit{fFork}(i))
\end{aligned}$$

The idea is to annotate all events in a process weak fair so that an enabled event of the process is not ignored forever. Model checking $\Box\Diamond \mathit{eat}.0$ against $\mathit{fCollege}(5)$ may return T0 and T2 as counterexamples but not T1. □

Example 3. The following specifies the Peterson's algorithm for mutual exclusion. Without fairness assumptions, the algorithm allows unbounded overtaking, i.e., a process which intends to enter the critical section may be overtaken by other processes infinitely (refer to [1] for a concrete example).

$$\begin{aligned}
P(i,j) &= (\mathit{sf}(\mathit{pos}[i] := j) \rightarrow \mathit{wf}(\mathit{step}[j] := i) \rightarrow \\
&\qquad [\mathit{step}[j] \neq i \vee \forall\, k \mid k \neq i \bullet \mathit{pos}[k] < j] \bullet P(i, j+1)) \\
&\quad \lhd\, j < N \,\rhd\, (\mathit{wf}(\mathit{cs}.i) \rightarrow \mathit{wf}(\mathit{pos}[i] := 0) \rightarrow P(i,1)) \\
\mathit{Peterson}(N) &= \Big\|_{i=1}^{N} P(i,1)
\end{aligned}$$

where N is the number of processes and pos, step are two lists of integers (with initial value 0) of size $N-1$ and N respectively. Infinite overtaking is evidenced by showing that $\mathit{Peterson}(N) \not\models \Box(\mathit{pos}[i] > 0 \Rightarrow \Diamond \mathit{cs}.i)$. Once a process has indicated that its intention to enter the critical section (by setting $\mathit{pos}[i]$ and $\mathit{step}[j]$), the assignment $\mathit{pos}[i] := j$ may be enabled only repeatedly. This is because it depends on the condition $[\mathit{step}[j] \neq i \vee \forall k \mid k \neq i \bullet \mathit{pos}[k] < j]$. Because the assignment $\mathit{step}[j] := i$ is not synchronized or guarded, weak fairness is sufficient to guarantee it will be engaged once enabled. The weak fairness associated with $\mathit{cs}.i$ and $\mathit{pos}[i] := 0$ prevents the system from idling forever. Notice that this is not necessary if we assume that the system shall never idle forever unless it is deadlocked. The above model guarantees that $\mathit{Peterson}(N) \models \Box(\mathit{pos}[i] > 0 \Rightarrow \Diamond \mathit{cs}.i)$. □

In order to guarantee a system is completely strongly fair, communicating events or events guarded by conditions must be annotated with strong fairness, whereas weak fairness is sufficient for local actions which are not guarded. Weak/strong fairness annotation allows us to model event-based fairness flexibly. In practice, even stronger fairness may be necessary. One example of a fairness constraint which is very strong is the notion of accepting states in Büchi automata, i.e., the system must keep moving until entering at least one accepting state (and do that infinitely often). Other examples of stronger fairness include the compassion conditions [16]. In order to capture these fairness constraints, we introduce two additional fairness annotations, which have the capability of driving the system to reach certain point. The additional annotations relies on the concept of "readiness" so that system behaviors may be restricted by fairness assumptions which are associated with events that are not even enabled.

Definition 3 (Readiness). *Let P be a process. Let V be a valuation of the variables.*

$$
\begin{array}{lll}
ready(Stop, V) & = ready(Skip, V) = \varnothing & \\
ready(e \rightarrow P, V) & = \{e\} & \\
ready(Skip;\ Q, V) & = ready(Q, V) & \\
ready(P;\ Q, V) & = ready(P, V) & - \textit{if } P \neq Skip. \\
ready(P \,\Box\, Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \sqcap Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \,\triangle\, Q, V) & = ready(P, V) \cup ready(Q, V) & \\
ready(P \lhd b \rhd Q, V) & = ready(P, V) & - \textit{if } V \models b. \\
ready(P \lhd b \rhd Q, V) & = ready(Q, V) & - \textit{if } V \models \neg\, b. \\
ready([b] \bullet P, V) & = ready(P, V) & - \textit{if } V \models b. \\
ready([b] \bullet P, V) & = \varnothing & - \textit{if } V \not\models b. \\
ready(P \,|[\, X \,]|\, Q, V) & = ready(P, V) \cup ready(Q, V) &
\end{array}
$$

Event e is *ready* given process P and valuation V if and only if $e \in ready(P, V)$. Note that *enabledness* and *readiness* are similarly defined for all process expressions except parallel composition. The difference is captured by the last line of the above definition. Given process $P \,|[\, X \,]|\, Q$, an event in X is enabled if and only if it is enabled in both P and Q, whereas it is ready if it is ready in either P or Q. Intuitively, an event is *ready* if and only if one thread of control is ready to engage in it. An *enabled* event must be *ready*. A *weak* live event, written as $wl(e)$, must be engaged if it is always ready (not necessarily enabled). Similarly, a *strong* live event, written as $sl(e)$, must be engaged if it is repeated ready[1]. Because whether an event is ready or not depends on only one process (in a parallel composition), live events may be used to design a controller which drives the execution of a given system.

Example 4. Let $LiftSystem$ be the modeling of a multi-lift system, which contains two events $turn\_on\_light$ and $turn\_off\_light$. In order to model that the light is always eventually turned off, the $LiftSystem$ may be replaced by $LightSystem \parallel LightCon$ where $LightCon = turn\_on\_light \rightarrow wl(turn\_off\_light) \rightarrow LightCon$. Because

[1] A similar modeling concept is hot locations in Live Sequence Charts [8], which force the system to move beyond.

both events must be synchronized, whenever event $turn\_on\_light$ is engaged, event $turn\_off\_light$ becomes ready. In this case, it remains ready until it is engaged. Thus, by definition, the light must eventually be turned off. □

Example 5. With live events, the dining philosophers may be modified as follows,

$$\begin{aligned} lPhil(i) &= wl(get.i.(i+1)\%N) \rightarrow get.i.i \rightarrow eat.i \\ &\quad \rightarrow put.i.(i+1)\%N \rightarrow put.i.i \rightarrow lPhil(i) \\ lFork(i) &= get.i.i \rightarrow wl(put.i.i) \rightarrow lFork(i) \ \Box \\ &\quad get.(i-1)\%N.i \rightarrow wl(put.(i-1)\%N.i) \rightarrow lFork(i) \\ lCollege(N) &= \Big\|_{i=1}^{N-1} (lPhil(i) \parallel lFork(i)) \end{aligned}$$

Model checking $\Box\Diamond eat.0$ against $lCollege(5)$ returns true. Initially, $wl(get.i.(i+1)\%N)$ is ready and therefore by definition, it must be engaged (since it is not possible to make it not ready). Once $get.i.(i+1)\%N$ is engaged, $wl(put.(i-1)\%N.i)$ becomes ready and thus the system is forced to execute until it is engaged. For the same reason, $wl(put.i.i)$ must be engaged afterwards. Once $put.i.i$ is engaged, $wl(get.i.(i+1)\%N)$ becomes ready again. Therefore, the system is forced to execute infinitely and fairly. The traces which lead to the deadlock state is not returned as a counterexample. This is because event $wl(put.(i-1)\%N.i)$ is ready in the deadlock state. Hence the trace is considered invalid because it does not satisfied the fairness assumption, i.e., the event $wl(put.(i-1)\%N.i)$ is always ready but never engaged. Refer to Example 6 for further explanation. □

The fairness annotations restrict the possible behaviors of the system. It thus results in a smaller set of traces. Note that fairness constraints cannot be captured using structural operational semantics. Therefore, a two-levels semantics is used to prune un-fair traces from infinite traces. Let Σ_{wf}, Σ_{sf}, Σ_{wl} and Σ_{sl} be the set of all weak fair, strong fair, weak live and strong live events respectively.

Definition 4 (Fair Traces). *Let P be a process. Let V be a valuation of the data variables. The set of fair traces is written as $ftraces(P, V)$. An infinite sequence of events $\tilde{tr} : \Sigma^\omega$ is in $ftraces(P, V)$ if and only if there exists an infinite sequence of $\tilde{P}$ and $\tilde{V}$ such that*

- *$\tilde{tr}$ is in $inftr(P, V)$,*
- *for all i, if there exists $e : \Sigma_{wf}$ such that e is* enabled *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or e is not* enabled *at state $(\tilde{P}(j), \tilde{V}(j))$.*
- *for all i, if there exists $e : \Sigma_{sf}$ such that e is* enabled *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or for all k such that $k \geq j$, e is not* enabled *at state $(\tilde{P}(k), \tilde{V}(k))$.*
- *for all i, if there exists $e : \Sigma_{wl}$ such that e is* ready *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or e is not* ready *at state $(\tilde{P}(j), \tilde{V}(j))$.*
- *for all i, if there exists $e : \Sigma_{sl}$ such that e is* ready *at state $(\tilde{P}(i), \tilde{V}(i))$, there exists j such that $j \geq i$ and $\tilde{tr}(j) = e$ or for all k such that $k \geq j$, e is not* ready *at state $(\tilde{P}(k), \tilde{V}(k))$.*

Compared to Definition 1, the additional constraint states that an infinite trace must be *fair*. That is, whenever a weak fair (live) event is enabled (ready), later it must be either engaged or become not enabled (ready); whenever a strong fair (live) event is enabled (ready), either it becomes not enabled (ready) forever after some execution or it is eventually engaged. By definition, all traces in $ftrace(P, V)$ satisfy the fairness constraints regarding the annotated events (see proof in [28]). Compared with previous proposals [6, 7, 3], our notion of fair events is more flexible and natural. For example, in [3] fairness constraints only concern parallel composition, whereas in our setting fairness concerns individual events and thus not only parallel composition but also choice and others. For instance, the process $P = sl(a) \rightarrow b \rightarrow P \,\Box\, sl(b) \rightarrow a \rightarrow P$ requires that the choice must not be completely biased, i.e., both choices must eventually be taken.

CSP algebraic laws [14] are largely preserved in our extended semantics, e.g., the symmetry and associativity laws of parallel composition. Nevertheless, a few do not apply any more because of the weak/strong live events, e.g., a new expansion laws for parallel composition is needed (refer to [28]). Moreover, the fairness might be overwhelming so that the specification may become infeasible. For instance, given $P = wl(e) \rightarrow P$, $ftraces(P \parallel (e \rightarrow College(5)), \varnothing) = \varnothing$. This specification is not feasible because the fairness constraint can never be satisfied, i.e., event e can be engaged only once. This boils down to the question on how to effectively verify a model under the embedded fairness.

4 Verification

In this section, we show that existing state-of-the-art model checking algorithms may be extended to handle our notion of event-based fairness with little computational overhead. We define the notion of feasibility and then present an algorithm for feasibility checking. A specification is feasible if it allows at least one infinite trace. Given a process and a valuation of the data variables, a feasibility checking checks whether there exists an infinite trace which satisfies the fairness constraints. The same algorithm is used for LTL model checking. The product of the model and the Büchi automaton generated from negation of the property is feasible if only and if the property is not true. For simplicity, we assume the number of system states is always finite, i.e., the domains of the variables are finite and the process specifies regular languages.

Definition 5 (State Graph). *Let P_0 be a process. Let V_0 be the valuation of the variables. A state graph $G_{(P_0, V_0)}$ is (S, s_0, E) where S is a set of system states of the form (e, P, V); s_0 is the initial state $(init, P_0, V_0)$ where $init$ is the event of system initialization; and E is a set of edges such that $((e, P, V), (e', P', V')) \in E \Leftrightarrow (P, V) \stackrel{e'}{\Rightarrow} (P', V')$.*

Without loss of generality, the just-engaged events are stored as part of the state information instead of transition labels, which turns a labeled transition system to a directed graph. A run of $G_{(P,V)}$ is an infinite sequence of vertices following the edges. It is straightforward to show that for all tr such that $tr \in inftr(P, V)$ if and only if there is a corresponding run in $G_{(P,V)}$. There is a loop in $G_{(P,V)}$ if and only if a vertex is reachable from the initial state and itself.

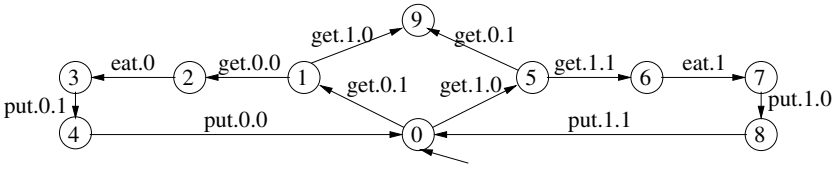


Fig. 1. LTS for 2 Dining Philosophers

Definition 6 (Fair Loop). *Let P_0 be a process. Let V_0 be a valuation of the data variables. Let $\langle (a_i, P_i, V_i), (a_{i+1}, P_{i+1}, V_{i+1}), \cdots, (a_j, P_j, V_j), (a_i, P_i, V_i)\rangle$ where $j \geq i$ be a loop in $G_{(P_0,V_0)}$. Let $Engaged = \{a_k \mid i \leq k \leq j\}$ be the set of engaged events during the loop. The loop is* fair *if and only if the following are satisfied,*

- $\bigcap_{k=i}^{j}(enabled(P_k, V_k) \cap \Sigma_{wf}) \subseteq Engaged$
- $\bigcup_{k=i}^{j}(enabled(P_k, V_k) \cap \Sigma_{sf}) \subseteq Engaged$
- $\bigcap_{k=i}^{j}(ready(P_k, V_k) \cap \Sigma_{wl}) \subseteq Engaged$
- $\bigcup_{k=i}^{j}(ready(P_k, V_k) \cap \Sigma_{sl}) \subseteq Engaged$

The set $\bigcap_{k=i}^{j}(enabled(P_k, V_k) \cap \Sigma_{wf})$ contains the weak fair events which are always enabled during the loop. Similarly, $\bigcap_{k=i}^{j}(ready(P_k, V_k) \cap \Sigma_{wf})$ is the set of weak live events which are always ready during the loop. The set $\bigcup_{k=i}^{j}(enabled(P_k, V_k) \cap \Sigma_{sf})$ contains the strong fair events which are enabled once during the loop. Similarly, $\bigcup_{k=i}^{j}(ready(P_k, V_k) \cap \Sigma_{sl})$ is the set of strong live events which are ready once during the loop. A loop is fair if and only if,

- all always-enabled weak fair events are engaged,
- all once-enabled strong fair events are engaged,
- all always-ready weak live events are engaged,
- all once-enabled strong live events are engaged.

A loop may contain the same state more than once, e.g., states 0,1,2,3,4,0,5,6,7,8,0 in Figure 1 forms a loop. It is straightforward to prove that a specification is feasible if and only if the state graph contains a fair loop. Feasibility checking is thus reduced to find a fair loop if possible. Equivalently, we can show that a specification is feasible if and only if the graph contains a fair strongly connected component (SCC) [16]. A strongly connected subgraph is fair if and only if the loop which visits every vertex in the subgraph is fair.

Example 6. Figure 1 shows the labeled transition system generated from $lCollege(2)$ (presented in Example 5). The loop containing state 0,1,2,3,4 is not fair because event $get.1.0$, which is annotated weak live, is always ready during the loop (though not always enabled, Definition 3). Similarly, the loop containing states 0,5,6,7,8 is not fair neither because $get.0.1$ is always ready. Note that the deadlock state 9 is considered as a trivial loop. It is not fair because both $put.0.1$ and $put.1.0$, which are annotated weak live, are ready (and therefore trivially always ready during the loop). The loop containing 0,1,2,3,4,5,6,7,8 (which constitute an SCC) is fair. Note that this loop satisfies the property $\Box\Diamond eat.0$. □

```
1. preorder, lowlink, found := ∅; stack; done := 1; i := 0;
2. working := ⟨(Init, P0, V0, S0)⟩;
3. while working ≠ ⟨⟩
4.       v = (a, P, V, S) := working.peek();
5.       if preorder(v) = null then preorder[v] := i++;
6.       foreach v' ∈ ample(P, V, S)
7.             if preorder(v') = null then working.push(v'); done := 0; break;
8.             else early-fair-loop-detection
9.       if done = 1
10.            lowlink[v] := preorder[v];
11.            foreach w ∈ ample(P, V, S)
12.                  if w ∉ found
13.                        if preorder[w] > preorder[v]
14.                              lowlink[v] := min(lowlink[v], lowlink[w]);
15.                        else lowlink[v] := min(lowlink[v], preorder[w]);
16.                  working.pop();
17.                  if lowlink[v] = preorder[v]
18.                        found.add(v); scc := {v};
19.                        while stack ≠ ⟨⟩ ∧ preorder[stack.peek()] > preorder[v]
20.                              k := stack.pop(); found.add(k); scc.add(k);
21.                        if scc is Büchi-fair
22.                              if scc is fair and nontrivial then return false;
23.                              if not OntheflyMC2(scc \ bad(scc)) then return false;
24.                  else stack.push(v);
25. return true;
```

Fig. 2. On-the-fly Model Checking Algorithm: $OnTheFlyMC1$

4.1 On-the-Fly Verification

In literature, there are two sets of algorithms for identifying a loop or equivalently checking the emptiness of Büchi automata, namely, ones based on nested depth-first-search and ones based on SCC (refer to the survey in [26]). We present here an SCC-based algorithm which extends the one presented in [12].

The problem of LTL model checking with event-based fairness is to verify whether every fair trace of the model satisfies a given LTL formula ϕ. Or equivalently, let $B_{\neg\phi}$ be a Büchi automaton which is equivalent to the negation of ϕ, the model violates ϕ if and only if there is a *fair* SCC in the synchronized product of $G_{(P,V)}$ and $B_{\neg\phi}$ which contains at least one accepting state from the Büchi automaton. In [16], a backward searching algorithm is used to identify all fair maximum SCCs if there is any. Because the query language of PAT is based on LTL (extended with events), we developed an on-the-fly approach based on Tarjan's algorithm for finding one maximum SCC. The idea is to search for maximum SCCs while building the state graph. If the found one is not fair, a set of 'bad' states are pruned and then the SCC is decomposed into smaller SCCs. Whenever a fair SCC is found, we proceed to produce a counterexample.

Figure 2 shows the detailed algorithm. The inputs are a process P_0, a valuation of the data variables V_0 and an initial state of the Büchi automaton S_0. Note that S_0 is

skipped for feasibility checking. The main loop from line 3 to 25 is based on an iterative improved version of Tarjan's algorithm (refer to [20, 12] for details). Stack $working$ holds all states that are yet to explored and $stack$ holds states which may be part of an SCC. At line 6 (and 11), a subset of the enabled actions (i.e., **ample**(P, V, S) which will be explained in detail in Section 4.2) is expanded. In order to conclude as soon as possible, a simple procedure is added at line 8 to check whether the found loop is fair. Experiences show that although this procedure becomes overhead for true properties, it may produce a counterexample early if there is any. This is particularly true for models which are strongly connected. A maximum SCC is discovered once the condition at line 17 is satisfied. Line 18 to 20 collect states of the SCC from *stack*. If the found SCC contains a Büchi accepting state (as a component of one state in scc), i.e., satisfying the condition at line 21, and if scc is fair and nontrivial (i.e., with at least one transition), the algorithm returns false after producing a counterexample (refer to algorithms presented in [16] on how to produce counterexamples). If the SCC is not fair, a set of bad states is removed from scc (line 23). A *bad* state carries a fairness constraint which can not be fulfilled by any loop formed by states in the SCC. A node (e, P, V) in scc is bad, i.e., $(e, P, V) \in bad(scc)$, if and only if one of the following conditions is satisfied,

- there exists $x \in \Sigma_{wf}$ such that $x \in enabled(P, V)$ and there does *not* exist a state (e', P', V') in scc such that $e' = x$ or $x \notin enabled(P', V')$. That is, x is *always* enabled but never engaged.
- there exists $x \in \Sigma_{sf}$ such that $x \in enabled(P, V)$ and there does *not* exist a state (x, P', V') in scc. That is, x is enabled but never engaged.
- there exists $x \in \Sigma_{wl}$ such that $x \in ready(P, V)$ and there does *not* exist a state (e', P', V') in scc such that $e' = x$ or $x \notin ready(P', V')$. That is, x is *always* ready but never engaged.
- there exists $x \in \Sigma_{sl}$ such that $x \in ready(P, V)$ and there does *not* exist a state (x, P', V') in scc. That is, x is ready in SCC but never engaged.

Then algorithm $OnTheFlyMC2$ is invoked at line 23. The logic of $OnTheFlyMC2$ is the same as $OnTheFlyMC1$ except that it only searches for maximum SCCs within the given states (and transitions which have been stored externally during $OnTheFlyMC1$). Refer to [28] for the details of $OnTheFlyMC2$. Whenever $OnTheFlyMC2$ returns false (i.e., a nontrivial fair SCC is found), we conclude a counterexample is found. If $OnTheFlyMC2$ returns true (i.e., no fair SCC is found) or there is a weak fair/live event which is always ready/enabled but never engaged or a Büchi fairness condition is not fulfilled by scc, scc is abandoned and then we proceed to search for the next maximum SCC. The soundness of Algorithm 2 is presented in Appendix.

Example 7. Applying feasibility checking to $lCollege(2)$ (shown in Figure 1) would return two maximum SCCs, i.e., one containing state 9 only and one containing the rest. By definition, the one containing 9 is not fair (as discussed) and state 9 is a bad state. Removing state 9 from the SCC results in an empty set and thus it is abandoned. The other SCC is fair and therefore the model is feasible. □

4.2 Partial Order Reduction

In the worst case, where the whole system is one SCC or the property is true, Algorithm 2 constructs the complete state graph and suffers from state space explosion. We thus apply partial order reduction to solve the problem. The idea is to only construct a *reduced* graph (in contrast to the complete graph in Definition 5) which is equivalent to the complete one with respect to the given property. This is realized by exploring only a subset of the enabled transitions at line 6 of algorithm 2.

Partial order reduction has been explored for almost two decades now. There have been theoretical works on partial order reduction under fairness constraints [22] in a different setting. In our setting, not only the reduction shall respect the property but also the fairness constraints. That is, a fair loop must be present in the reduced graph if there is one in the complete graph. In the *reduced* graph, for every node, only a subset of the enabled synchronized (outgoing) transition of the model and the Büchi automaton is explored. In particular, given a state (a, P, V, S) where a is the just engaged event, P is the current process expression, V is the current valuation and S is the current state of the Büchi automaton, a successor node (a', P', V', S') is explored, written as $(a', P', V', S') \in \textbf{ample}(P, V, S)$, if and only if the following conditions are satisfied,

- (a', P', V', S') is a successor in the complete graph, i.e., $(P, V) \stackrel{a'}{\Rightarrow} (P', V')$ and the transition is allowed by the Büchi automaton, i.e., (S, a', S') is a transition in $B_{\neg \phi}$ and the condition which guards the transition is true. Note that the Büchi automata are transition-labeled for efficient reasons.
- the successors must satisfy a set of additional conditions for property-preserving partial order reduction, which is denoted as $(P', V') \in ample(P, V)$.

The algorithm presented in Figure 3 has been implemented in PAT to produce small but sound *ample*(P, V), which extends the one proposed in [11] to handle event annotations.

If P is not an indexed parallel composition (or indexed interleaving), the node is fully expanded. Otherwise, we identify one process which satisfies a variety of conditions and expand the node with only enabled events from that process. Notice that $enabled_{P_i}(P, V)$ is $enabled(P, V) \cap enabled(P_i, V)$, e.g., the set of globally enabled events which P_i participates in. $current(P_i)$ is the set of actions that could be enabled given P_i and a cooperative environment. For instance, a guarded event is in the set even if the guard condition is false. Two events are dependent, written as $dep(e, e')$ if they synchronize or write/read a shared variable (with at least one writing). Note that both $current(P_i)$ and $dep(e, e')$ are based on static information, which can be collected during compilation. A component P_i is chosen if and only if the following conditions are satisfied,

- $enabled_{P_i}(P, V) = current(P_i)$. No other events could be enabled given P_i. Refer to [11] for intuitions behind this condition.
- The condition $on\_stack(P', V')$ is true if and only if the state (P', V') is on the search stack (refer to [11]). Performing any event in $enabled_{P_i}(P, V)$ must not result in a state on the search stack. This is to prevent enabled actions from being ignored forever. Note that this condition can be removed for checking deadlock-freeness.

1. **if** P is of the form $P_1 \parallel P_2 \parallel \cdots \parallel P_n$
1. **foreach** i such that $1 \leq i \leq n$
2. **if** $enabled_{P_i}(P, V) = current(P_i)$
3. $ample := true$;
4. $ampleset := \varnothing$;
5. **foreach** e, P', V' s.t. $e \in enabled_{P_i}(P, V)$ and $(P, V) \stackrel{e}{\Rightarrow} (P', V')$
6. **if** $\underline{visible(e)} \vee on\_stack(P', V') \vee \exists\, e' : \Sigma_{P_j} \mid i \neq j \bullet dep(e, e')$
7. $ample := false$; **break**;
8. **else**
9. $ampleset := ampleset \cup \{(e, P', V')\}$;
10. **endif**
11. **endfor**
12. **if** $ample$ **then return** $ampleset$;
13. **endif**
14. **endfor**
15. **endif**
16. **return** $\{(e, P', V') \mid e \in enabled(P, V)$ and $(P, V) \stackrel{e}{\Rightarrow} (P', V')\}$;

Fig. 3. Partial Order Reduction

- The actions in $enabled_{P_i}(P, V)$ must be independent from transitions of other components, i.e., $\nexists e' : \Sigma_{P_j} \mid i \neq j \bullet dep(e, e')$. Refer to [11] for intuitions behind this condition.
- Different from the one in [11], an event is visible, i.e., written as $visible(e)$, if it is visible to a given property or the fairness annotations. Event e is visible to a property if and only if e constitutes the property, e.g., $eat.0$ is visible given property $\Box\Diamond eat.0$, or e updates a variable which constitutes the property. Event e is visible to the fairness annotations if and only if performing e may change the set of annotated events which are enabled or ready.

The soundness of the partial order reduction with respect to next-free LTL and event-based fairness is proved in Appendix. Note that the above realizes only one of the possible heuristics for partial order reduction, which we believe is cost-effective. Achieving the maximum reduction is in general computational expensive.

Besides partial order reduction, we have also implemented optimizations based on CSP's algebraic laws. For instance, in order to handle systems with large number of identical or similar processes, efficient procedures are applied to sort the processes of a parallel or interleaving composition. The soundness is proved by the symmetry and associativity of indexed interleaving and parallel composition.

4.3 Experiments

In this part, we evaluate the algorithm and the effectiveness of the reductions using benchmarks systems. Table 2 presents a part of the experimental results. The experiments are conducted on Windows XP with a 2.0 GHz Intel CPU and 1 GB memory.

Table 2. Experiment Results

Model	Property	without fairness			with event-based fairness		
		result.	w/o red.	with red.	result.	w/o red.	with red.
$College(7)$	$\Box\Diamond eat0$	No	< 1	< 1	Yes	10.3	11.3
$College(9)$	$\Box\Diamond eat0$	No	< 1	< 1	Yes	469.1	504.4
$College(11)$	$\Box\Diamond eat0$	No	4.2	< 1	Yes	–	–
$College(13)$	$\Box\Diamond eat0$	No	25.6	< 1	Yes	–	–
$College(15)$	$\Box\Diamond eat0$	No	–	< 1	Yes	–	–
$Milner\_Cyclic(10)$	$\Box\Diamond work0$	Yes	17.8	< 1	Yes	17.7	< 1
$Milner\_Cyclic(12)$	$\Box\Diamond work0$	Yes	322.9	< 1	Yes	283.3	< 1
$Milner\_Cyclic(100)$	$\Box\Diamond work0$	Yes	–	3.3	Yes	–	3.4
$Milner\_Cyclic(200)$	$\Box\Diamond work0$	Yes	–	17.6	Yes	–	18.1
$Milner\_Cyclic(400)$	$\Box\Diamond work0$	Yes	–	118.4	Yes	–	119.2
$ReadersWriters(100)$	$\Box!error$	Yes	–	4.3	Yes	–	3.9
$ReadersWriters(200)$	$\Box!error$	Yes	–	37.3	Yes	–	29.1
$ReadersWriters(400)$	$\Box!error$	Yes	–	251.3	Yes	–	257.1

The first model is the dining philosophers. The property is $\Box\Diamond eat.0$. Without fairness assumptions, this property is false. A counterexample is quickly produced in most of the cases. Nonetheless, it may take considerably long if the trace leading to a counterexample is explored very late (e.g., for $College(15)$ without reduction). Partial order reduction significantly reduces the time to discover a counterexample for this example. This model is then annotated with fairness, as shown in Example 5. The last three columns show verification results of $lCollege(N)$. The property becomes true and therefore a complete search is necessary. Note that partial order reduction gains little. The reason is that the model is highly coupled and heavy in communication. Manually hiding local communicating could reduce the verification time, as shown in [24].

The second model is Milner's cyclic scheduler. Milner's cyclic scheduler describes a scheduler for N concurrent processes. The processes are scheduled in cyclic fashion so that the first process is reactivated after the N-th process has been activated. The fairness assumptions state that a process must eventually finish its local task and then activate the next process. The property to verify is that a process must eventually be scheduled, which is true with/without the fairness assumptions. This model demonstrates the effectiveness of the partial order reduction. Without the reduction, the size of the search graph grows exponentially and thus verification soon becomes infeasible (e.g., for 15 processes). With partial order reduction, we are able to verify 400 processes reasonably fast (using less than 2 minutes). Notice that the computational overhead for handling fairness annotations are negligible, e.g., same amount of time is taken to verify the model with/without fairness. The third models the classic readers/writers problem. The readers/writers model describes a protocol for coordination of N identical readers and N identical writers accessing a shared resource. The property to verify is reachability of an erroneous situation (i.e., wrong readers/writers coordination). This mode is then annotated with fairness assumptions to state that each reader/writer must eventually finish reading/writing. This model demonstrates the effectiveness of the reduction for handling identical/similar processes.

Details of the models and more experiments are available online [28]. Compared to existing tools, PAT complements the CSP model checker FDR in serval aspects. Namely, PAT supports verification under fairness assumptions and temporal logic based verification. For common features like deadlock-freeness checking, PAT outperforms FDR sometimes because we use a completely on-the-fly checking strategy (refer to the results at our web site). Compared to SPIN, PAT is not yet as efficient for systems with no event-based fairness and small LTL properties. PAT offers a more flexible way of modeling fairness and verifying under fairness assumptions (than SPIN's option for process-level weak fairness). PAT differs from SPIN in two aspects. Firstly, because we are dealing with an event-based formalism, we extend LTL with events so that properties concerning both states and events can be stated and verified. Secondly, because fairness constraints have been embedded in the specification, the size of the property is reduced and thus model checking under fairness is carried out efficiently.

5 Conclusion and Future Works

In this work, we presented an approach to systematically model a variety of fairness in event-based compositional systems. We also developed algorithms to efficiently verify systems under fairness assumptions. A toolset named PAT has been developed for specification and verification of event-based fairness enhanced systems. Our experiments show clear advantage over the common practise of assuming a fair scheduler and then proving liveness properties over safety-centric specifications.

As for future works, there are a number directions to go in terms of tool development. We are currently adding more language features, e.g., arrays, broadcasting messages, etc. We are exploring how to extend event-based fairness and its verification to languages like C# or Java. PAT is not yet as efficient for systems with no event-based fairness and small LTL properties. Optimization techniques like symmetry reduction need to be studied under fairness and incorporated. One future work of theoretical interests is to study the notion of process refinement/equivalence for fairness enhanced processes. Because of fairness constraints, trace refinement becomes a stronger notion. Namely, a fair branching may not be removed in a refined process without introducing new traces. In this work, we choose not to prevent inputs from being marked *fair*. Marking inputs from an open channel *fair* restricts the behaviors of the environment, which could be largely undesirable. Nevertheless, assuming fair/live environments would help effective model checking of open systems and synthesis (e.g., [27]). One future work is to investigate verification/synthesis of open systems under fairness assumptions.

Acknowledgement

Jun Pang, Yuxin Deng and anonymous referees provided helpful comments on early drafts of this paper. This work is partially supported by the research grant titled "Sensor Networks Specification and Validation" (T1 251RES0716) funded by Ministry of Education, Singapore.

References

1. Alagarsamy, K.: Some Myths About Famous Mutual Exclusion Algorithms. SIGACT News 34(3), 94–103 (2003)
2. Apt, K.R., Francez, N., Katz, S.: Appraising Fairness in Languages for Distributed Programming. Distributed Computing 2, 226–241 (1988)
3. Brookes, S.D.: Traces, Pomsets, Fairness and Full Abstraction for Communicating Processes. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 466–482. Springer, Heidelberg (2002)
4. Brookes, S.D., Roscoe, A.W., Walker, D.J.: An Operational Semantics for CSP. Technical report (1986)
5. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
6. Costa, G., Stirling, C.: Weak and Strong Fairness in CCS. In: Chytil, M.P., Koubek, V. (eds.) MFCS 1984. LNCS, vol. 176, pp. 245–254. Springer, Heidelberg (1984)
7. Costa, J.F., Sernadas, A.: Progress Assumption in Concurrent Systems. Formal Aspects of Computing 7(1), 18–36 (1995)
8. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
9. Dong, J.S., Hao, P., Sun, J., Zhang, X.: A Reasoning Method for Timed CSP Based on Constraint Solving. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 342–359. Springer, Heidelberg (2006)
10. Song Dong, J., Hao, P., Qin, S., Sun, J., Wang, Y.: Timed Patterns: TCOZ to Timed Automata. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 483–498. Springer, Heidelberg (2004)
11. Grumberg, O., Clarke, E.M., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
12. Geldenhuys, J., Valmari, A.: More efficient on-the-fly LTL verification with Tarjan's algorithm. Theoritical Computer Science 345(1), 60–82 (2005)
13. Henzinger, M.R., Telle, J.A.: Faster Algorithms for the Nonemptiness of Streett Automata and for Communication Protocol Pruning. In: Karlsson, R., Lingas, A. (eds.) SWAT 1996. LNCS, vol. 1097, pp. 16–27. Springer, Heidelberg (1996)
14. Hoare, C.A.R.: Communicating Sequential Processes. Inte. Series in Computer Science. Prentice-Hall, Englewood Cliffs (1985)
15. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engeering 23(5), 279–295 (1997)
16. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: Model Checking with Strong Fairness. Formal Methods and System Design 28(1), 57–84 (2006)
17. Lamport, L.: Fairness and Hyperfairness. Distributed Computing 13(4), 239–245 (2000)
18. Latvala, T., Heljanko, K.: Coping with Strong Fairness. Fundamenta Informaticae 43(1–4), 175–193 (2000)
19. Liu, Y., Sun, J., Dong, J.S.: An Analyzer for Extended Compositional Process Algebras. In: 30th International Conference on Software Engineering (ICSE 2008) Companion Volume, pp. 919–920. ACM Press, New York (2008)
20. Nuutila, E., Soisalon-Soininen, E.: On Finding the Strongly Connected Components in a Directed Graph. Information Processing Letters 49(1), 9–14 (1994)
21. Older, S.: Strong Fairness and Full Abstraction for Communicating Processes. Information and Computation 163(2), 471–509 (2000)
22. Peled, D.: Ten Years of Partial Order Reduction. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 17–28. Springer, Heidelberg (1998)

23. Puhakka, A., Valmari, A.: Liveness and Fairness in Process-Algebraic Verification. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 202–217. Springer, Heidelberg (2001)
24. 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)
25. Schneider, S.: Concurrent and Real-time Systems: the CSP Approach. John Wiley, Chichester (2000)
26. Schwoon, S., Esparza, J.: A Note on On-the-Fly Verification Algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
27. Sun, J., Dong, J.S.: Design Synthesis from Interaction and State-Based Specifications. IEEE Transactions on Software Engineering 32(6), 349–364 (2006)
28. Sun, J., Liu, Y., Dong, J.S., Wang, H.: The Process Analysis Toolset Pat. Technical report, `http://www.comp.nus.edu.sg/~sunj/pat.pdf`

Appendix: Soundness Proofs

Theorem 1. *Let P be a process and V be a valuation of the variables. Let ϕ be a next-free LTL formula (with events). $(P, V) \vDash \phi$ if and only if Algorithm 2 returns true.*

Proof. By a standard proof we can show that $(P, V) \vDash \phi$ if and only if there does not exist an infinite path of $G_{(P,V)} \times B_{\neg\phi}$ which is fair with respect to $G_{(P,V)}$ and is accepting to $B_{\neg\phi}$. Equivalently, $(P, V) \nvDash \phi$ if and only if there is a fair loop (since we assume $G_{(P,V)}$ is finite) in $G_{(P,V)} \times B_{\neg\phi}$ which is also accepting. Equivalently, $(P, V) \nvDash \phi$ if and only if there is a fair SCC which is also accepting to $B_{\neg\phi}$. To prove the above theorem, we thus need to show that if the algorithm returns false if and only if there is such an SCC. If there exists such an SCC (say scc), there must be one maximum SCC (say SCC) which contains scc. By the soundness of Tarjan's algorithm, SCC must be discovered by line 21. If scc is SCC, the algorithm returns false as we shall prove. Otherwise, because scc contains no bad states (by Definition 6 and the definition of bad states on page 13), all states of scc are not pruned. By induction we conclude either a fair and accepting SCC which contains all states of scc is found or scc is found. In both cases, the algorithm returns false. Thus, the algorithm returns false if there is a fair SCC which is also accepting. Ergo, it returns false if $(P, V) \nvDash \phi$. It is straightforward to prove that the algorithm returns false, either at line 8 (a fair loop which is also accepting is found) or line 22 (the maximum SCC found is fair and accepting) or line 23 (a sub-SCC is), only if such an SCC is found. The algorithm is terminating because the number of $States$ is finite (by assumption) and i is monotonically increasing (i.e., the number of visited states). The 'recursive' call at line 23 is terminating because the number of states in scc is monotonically decreasing (i.e., $bad(scc)$ is not empty by definition). Therefore, we conclude the algorithm returns true only if $(P, V) \vDash \phi$. □

Theorem 2. *Let P be a process. Let V be the valuation of the global variables. Let $R_{(P,V)}$ be the reduced graph constructed by expanding each node with only events returned by the Algorithm (shown in Figure 3). Then, for every next-free LTL formula ϕ, $G_{(P,V)}, s_0 \vDash \phi$ if and only if $R_{(P,V)}, s_0 \vDash \phi$.*

Proof. For simplicity, we only prove the case for weak live events, i.e., given the weak live annotations, there is a fair loop in the reduced graph if and only if there is one in the complete graph. Strong live events can be transformed to weak live events at the cost of auxiliary variables as shown in [16]. Weak/strong fair events can be proved similarly.

For each weak live event $wl(a)$, we introduce an auxiliary variable x_a. P is modified to be P' in which x_a is set to 0 if it is not ready or just engaged or otherwise set to 1. ϕ is then modified to be ϕ' which is of the form $(\wedge_a \ \Box\Diamond x_a = 0) \Rightarrow \phi$ for each auxiliary variable x_a. We show that $(P, V) \vDash \phi$ if and only if $(P', V) \vDash \phi'$. For every loop in the given model, if it is fair with respect to the fairness constraints, then for every auxiliary variable x_a, the loop satisfies that $\Box\Diamond x_a = 0$ because $wl(a)$ can not be always ready during the loop and never engaged by Definition 6. Thus, if the fair loop satisfies ϕ, it satisfies ϕ'. The reverse is proved trivially. Thus, $(P, V) \vDash \phi$ if and only if $(P', V) \vDash \phi'$.

Next, we apply Theorem 12 in Chapter 10 of [11] to show that $G_{(P', V)}, s_0 \vDash \phi'$ if and only if $R_{(P', V)}, s_0 \vDash \phi'$. By apply similar arguments, we can show that the Algorithm satisfies the following conditions,

C0. $ample(P, V)$ is empty if and only if $enabled(P, V)$ is empty. This is trivial.
C1. Along every path in the full state graph that starts at s, a transition that is dependent on a transition in $ample(P, V)$ cannot be executed without a transition in $ample(s)$ occurring first. This is proved by the same argument in Section 10.5.2 in [11].
C2. If a node is not fully expanded, then every event in $ample(P, V)$ is invisible. This is guaranteed by line 6 and 7 in the algorithm presented in Figure 3, i.e., the condition $visible$, so that only events which preserves the valuation of propositions in ϕ and the auxiliary variables are presented in the ample set .
C3. There must be at least one node which is fully expanded along a cycle. This is guaranteed by condition $on\_stack(P', V')$ at line 6.

Thus, we conclude that $G_{(P', V)}, s_0 \vDash \phi'$ if and only if $R_{(P', V)}, s_0 \vDash \phi'$. By transitivity, we conclude that the theorem holds. □

Modelling and Proof of a Tree-Structured File System in Event-B and Rodin*

Kriangsak Damchoom[1], Michael Butler[1], and Jean-Raymond Abrial[2]

[1] University of Southampton
United Kingdom
{kd06r,mjb}@ecs.soton.ac.uk
[2] ETH Zurich
Switzerland
jabrial@inf.ethz.ch

Abstract. Event-B is a formalism used for specifying and reasoning about complex discrete systems. The Rodin platform is a new tool for specification, refinement and proof in Event-B. In this paper, we present a verified model of a tree-structured file system which was carried out using Event-B and the Rodin platform. The model is focused on basic functionalities affecting the tree structure including create, copy, delete and move. This work is aimed at constructing a clear and accurate model with all proof obligations discharged. While constructing the model of a file system, we begin with an abstract model of a file system and subsequently refine it by adding more details through refinement steps. We have found that careful formulation of invariants and useful theorems that can be reused for discharging similar proof obligations make models simpler and easier to prove.

Keywords: File system, Tree structure, Refinement, Proof, Event-B, Rodin tool.

1 Introduction

Nowadays, there are many formal methods used in the area of software development together with a number of advanced theories and tools. However, more experiments in this area are still needed to be carried out in order to provide significant evidence for convincing and encouraging other users to benefit from those theories and tools, and make formal methods more accessible to software industries. We see our work as a contribution to the filestore mini-challenge proposed by Joshi and Holzmann [13]. As highlighted in [13], a filestore is a complex system that presents interesting challenges for specification and verification. For example, how do we ensure reliability in the presence of concurrent accesses or how do we deal with accidental failures that may occur during the execution.

* This work was part of the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) www.deploy-project.eu.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 25–44, 2008.

The file system is chosen as a case study for our experiment which is carried out by using Event-B [3] and the Rodin platform [8, 2] for specification, refinement and proof.

We make strong use of refinement to introduce gradually features to the formal model. We see the contribution as twofold. Firstly our work provides evidence of the applicability of the Event-B language, the refinement approach and of the Rodin tool. Secondly our experiment provides guidance on effective modelling and proof styles that may be of benefit to others working on formal development of similar systems.

Our specification of this system is focused on a tree structure and basic functionalities affecting the tree structure: create, delete, copy and move objects that can be files or directories. In this specification, we start with an abstract level accompanied with careful formulation of invariants, and then follow this by refinements in which more details are added.

In the abstraction, we introduce the two main properties of a tree structure: (i) there are no loops in a tree structure and (ii) every node in a tree is reachable from the root. For the no-loop property, instead of using transitive closure which is generally used for specification of absence of loops, we employ a no-loop property proposed in [3] to formulate a simpler invariant satisfying this property. Employing this property, which is less complicated than transitive closure, makes the model easier to prove. For the second property, reachability, instead of introducing another invariant, we introduce a machine theorem – which is derived from existing invariants – that can be proved to show that the property is satisfied.

In the first refinement, files and directories are introduced (in the previous abstraction, both files and directories are treated in a similar way as objects). Therefore, in this level, some additional invariants are added to the model. For instance, files and directories are distinct and each object's parent must be a directory.

In addition, other required properties, i.e., a content of file and access permissions, are introduced accompanied with related events concerning these properties in the second and the third refinement, respectively. Some constraints are covered in these two refinements, such as, each file has a content, each object has an owner and its permissions, accessing to each object is dependent on the permissions allowed, etc.

In total 162 proof obligations were automatically generated by the Rodin platform. 78% of them are proved automatically while others are discharged by using the interactive prover. Based on interactive proof, we introduced some proved theorems that can be reused for discharging several similar proof obligations. This makes interactive proof easier. Consequently, the time required was also reduced. An archive of our development in the Rodin tool may be downloaded [1]. This can be imported by anyone with an installation of the tool which is freely available [2].

[1] http://deploy-eprints.ecs.soton.ac.uk/22/
[2] www.event-b.org

This paper will begin with providing a short description of Event-B and the Rodin platform. Secondly, an informal description of a tree-structured file system and its constraints are given in Section 3. Thirdly, an Event-B specification of the file system which is divided into four levels (an abstraction and three levels of refinements) will be outlined in Section 4, 5, 6, 7 and 8. Fourthly, in Section 9, proof statistics will be figured. Finally, comparison with related work and conclusion will be given in Section 10 and 11 respectively.

2 Event-B and the Rodin Platform

Event-B [3] is an extension of the B-method [1] for specifying and reasoning about complex systems including concurrent and reactive systems. An Event-B model is described in terms of contexts and machines, see Fig. 1.

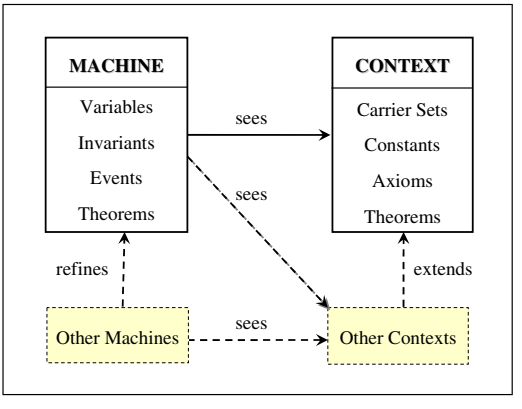


Fig. 1. Relationship between machines and contexts

Contexts [4, 5] contain the static parts of a model. Each context may consist of carrier sets and constants as well as axioms which are used to describe the properties of those sets and constants. Contexts may contain theorems for which it must be proved that they follow from the preceding axioms and theorems. Moreover, contexts can be extended by other contexts and seen by more than one machine. Additionally, a context may be indirectly seen by machines. Namely, a context C can be seen by a machine M indirectly if the machine M explicitly sees another context which is an extension of the context C.

Machines [4, 5] contain the dynamic parts of an Event-B model. This part is used to provide behavioural properties of the model. A machine is made of a state, which is defined by means of variables, invariants, events and theorems shown in Fig. 1. The theorems of a machine must be shown to follow from the context and the invariants of that machine. In addition, machines can be refined by other machines, but each machine can refine at most one machine.

Variables, like constants, correspond to mathematical objects: sets, binary relations, functions, numbers, etc. They are constrained by invariants $I(v)$ where v are the variables of the machine. Invariants are supposed to hold whenever variable values change. But this must be proved through the discharge of proof obligations [4].

A machine contains a number of atomic events which show the way that the machine may evolve. Each event is composed of three elements: an event name, guard(s) and action(s). The guard is the necessary condition for the event. The actions determine the way in which the state variables are going to evolve when performing the event [4].

An event is guarded and atomic and may be performed only when its guard holds. This means that when the guards of several events hold at the same time, then only one of them may be performed at that time. The event is non-deterministically chosen to be performed. Generally, an event, named Evt, is presented in one of three possible forms shown in Fig. 2. Where $S(v)$ are generalized substitutions of variable v, $G(v)$ represents a guard of event Evt, and t is a local variable [4].

Evt $\widehat{=}$ begin *S(v)* end
Evt $\widehat{=}$ when *G(v)* then *S(v)* end
Evt $\widehat{=}$ any t where *G(t,v)* then *S(t,v)* end

Fig. 2. Three posible forms of an event

The Rodin platform [8, 2] is an open and extensible tool for Event-B specification and verification. It contains a database of modelling elements used for constructing system models such as variables, invariants and events. It is accompanied by various useful plug-ins such as a proof-obligation generator, provers, model-checkers, UML transformers, etc [6].

3 An Informal Description of a Tree-Structured File System and Constraints

A tree-structured file system can be described in terms of a collection of objects representing files and directories and a set of operations that may be performed on these objects. The objects are structured as a tree. The tree has only one root directory that cannot be deleted, copied or moved. Each object except the root has only one parent which is a directory. Four operations affecting the tree structure are discussed below.

Create. Create an object in an existing directory. The object can be either a file or a directory.

Copy. Copy an existing object from one place to another place. The destination must exist and must not be a descendant of the object being copied or the

object itself. If the object being copied is a directory, all objects belong to that directory must also be copied to the new location and the copy must have the same structure as the original.

Move. Move an existing object in the tree structure from one place to another place. The destination must exist and must not be a descendant of the object being moved or the object itself.

Delete. Delete an existing object in the file system. In case of deleting a directory, all its descendants must also be removed.

4 Abstract Model

In this abstraction, we begin with an abstract model of a tree-structured file system focusing on tree properties and operations affecting the tree structure. However, files and directories are not distinguished in this level. Instead they are postponed to next refinement given in Section 6. Thus, in this level, both of them are treated in the same way as objects which are nodes of the tree structure. Below is a list of requirements in this level.

Req1.1: The tree has a root node.
Req1.2: All objects except the root node must have a parent.
Req1.3: There are no loops in the tree.
Req1.4: Every node in the tree is reachable from the root node.

Machine variables, invariants formulated to satisfy those required properties mentioned above, and initialised values of each variable are given in Fig. 3. These variables, invariants and initialisation are discussed below.

Variables $objects$, $parent$
Invariants
$inv1.1 : objects \subseteq OBJECT$
$inv1.2 : root \in objects$
$inv1.3 : parent \in objects \setminus \{root\} \rightarrow objects$
$inv1.4 : \forall s \cdot (s \subseteq parent^{-1}[s] \Rightarrow s = \varnothing)$
Initialisation
$objects := \{root\}$
$parent := \varnothing$

Fig. 3. Machine variables, invariants and initialisation of an abstract model

In a context seen by this abstract machine, $OBJECT$ is defined as a carrier set and $root$ is an $OBJECT$ constant (see Fig. 5). Considering Fig. 3, there are two state variables introduced in the machine: (i) $objects$, a set of existing

objects in the file system ($inv1.1$); and (ii) *parent*, a total function mapped from all objects except *root* to their parent which is an object. In this abstraction, *objects* and *parent* are initialised to a set consisting of *root* and the empty set respectively. Invariant $inv1.3$ states that all objects except *root* must have a parent. This invariant satisfies $Req1.2$. Invariant $inv1.4$ is introduced to ensure that there are no loops in the tree structure (satisfying $Req1.3$). This invariant is formulated by using the no-loop property proposed by Abrial in [3]. The reason we choose this formulation, instead of transitive closure which is generally used to specify tree properties – such as a specification of visual file system in [12] – is to make the model simpler and easier to prove.

Considering $inv1.4$, $parent^{-1}[s]$ gives the direct descendants of all elements of set s. For $s \subseteq objects, s \subseteq parent^{-1}[s]$ means that s contains a loop in the parent relationship. Hence, this invariant states that the only such set that can exist is the empty set and thus the parent structure cannot have loops. If we were to use transitive closure, we would need to add the property $inv1.4b$ given in Fig. 4 to the machine invariants.

$$inv1.4b : tcl(parent) \cap id(OBJECT) = \varnothing$$

Fig. 4. No-loop property

Here *tcl* which is mentioned in Invariant $inv1.4b$ is a transitive closure. In a context shown in Fig. 5, *tcl* is defined as a total function mapped from $OBJECT \leftrightarrow OBJECT$ to $OBJECT \leftrightarrow OBJECT$. Giving $r \in OBJECT \leftrightarrow OBJECT$, the transitive closure of r is equal to $r \cup r; tcl(r)$ ($thm1$ of Fig. 5).

The *parent* variable is updated by several of the events. If we were to use $inv1.4b$ instead of $inv1.4$, the *Copy* event, for example, would give rise to a proof obligation with $inv1.4b$ as a hypothesis and the following goal:

$$tcl(parent \cup replica \cup \{nobj \mapsto to\}) \cap id(OBJECT) = \varnothing$$

Proof of this PO would not be easy since distribution of *tcl* through union and other set operations is not straightforward. We avoid such difficulty proofs by using formulation $inv1.4$ instead. Significantly, we can prove that the formulation in $inv1.4b$ follows from the formulation in $inv1.4$. This is given by Theorem $thm3$ shown in Fig. 5. This theorem has been proved using the interactive prover of Rodin. The strategy we follow in proving this theorem is to use proof by contradiction.

In order to satisfy requirement $Req1.4$, instead of introducing another invariant, we present other machine theorems (given in Fig. 6) which are derived from existing invariants and guarantee that the property is satisfied. Considering Theorem $mth3$, since $(tcl(parent))^{-1}[\{root\}]$ returns all objects reachable from *root*, this theorem shows that all objects except *root* are reachable from *root*. Other machine theorems, $mth1$ and $mth2$, are used in the proof of $mth3$. Theorem $mth4$ is introduced to satisfy the no-loop property.

Sets
$OBJECT$
Constants
$root, tcl, objrel, objfn$
Axioms
$axm1 : root \in OBJECT$
$axm2 : objrel = OBJECT \leftrightarrow OBJECT$
$axm3 : objfn = OBJECT \setminus \{root\} \nrightarrow OBJECT$
$axm4 : tcl \in objrel \rightarrow objrel$
$axm5 : \forall r \cdot (r \in objrel \;\Rightarrow\; r \subseteq tcl(r))$
$axm6 : \forall r \cdot (r \in objrel \;\Rightarrow\; r;tcl(r) \subseteq tcl(r))$
$axm7 : \forall r, t \cdot (r \in objrel \;\wedge\; r \subseteq t \;\wedge\; r;t \subseteq t \;\Rightarrow\; tcl(r) \subseteq t)$
Theorems
$thm1 : \forall r \cdot (r \in objrel \;\Rightarrow\; tcl(r) = r \cup (r;tcl(r)))$
$thm2 : tcl(\varnothing) = \varnothing$
$thm3 : \forall t \cdot (t \in objfn \;\wedge\; (\forall s \cdot s \subseteq (t^{-1})[s] \;\Rightarrow\; s = \varnothing)$
$\Rightarrow\; tcl(t) \cap id(OBJECT) = \varnothing)$

Fig. 5. Definition of transitive closure (tcl) and no-loop theorem ($thm3$)

Theorems
$mth1 : \forall T \cdot (root \in T \;\wedge\; parent^{-1}[T] \subseteq T \;\Rightarrow\; objects \subseteq T)$
$mth2 : objects \subseteq \{root\} \cup (tcl(parent))^{-1}[\{root\}]$
$mth3 : objects \setminus \{root\} \subseteq (tcl(parent))^{-1}[\{root\}]$
$mth4 : tcl(parent) \cap id(OBJECT) = \varnothing$

Fig. 6. Machine theorems satisfying reachability and no-loop properties

5 Events

In this section, we outline four abstract events including *Create*, *Move*, *Copy* and *Delete*.

Create event. Create an object in an existing location (see Fig. 7). In the figure, *obj* is an object being created and *in* is its parent. Here *obj* must be an OBJECT that is not already in the set *objects* (see *grd*1); and *in* must exist (see *grd*2). The object *obj* will be added to the set *objects* by *act*1; and *in* will be assigned to be the *obj*'s parent by *act*2.

Move event. This event is aimed at moving an existing object except *root* from one place to another place. Considering Fig. 8, a is an object being moved from node r to node c. Node c will become a new parent of a. In Fig. 9, an

Event Create $\widehat{=}$
  Any
    obj, in
  Where
    $grd1 : obj \in OBJECT \setminus objects$
    $grd2 : in \in objects$
  Then
    $act1 : objects := objects \cup \{obj\}$
    $act2 : parent(obj) := in$
  End

Fig. 7. A specification of Event *Create*

existing object named *obj* is moved to a new location named *to*. Parameter *des* is the set of all descendants of *obj* which is equal to $(tcl(parent))^{-1}[\{obj\}]$. In this case, the destination, *to*, must exist and not be *obj* or a descendant of *obj* (these constraints are specified as $grd2$ and $grd5$). These guards are necessary to guarantee that the move does not introduce a loop or unreachable objects. The *parent* function is updated so that *obj* has *to* as its parent.

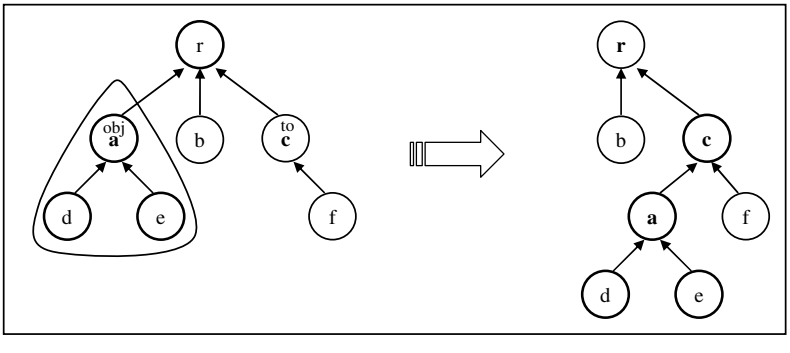


Fig. 8. Diagram of moving a subtree rooted at a from r to c

Copy event. In order to understand more about the copy event, we will describe this event by using Fig. 10. From the figure, the left-hand side is a tree before copying and the right-hand side is the result. Here r is a root node, a is an object being copied (d and e, its descendants, will be copied as well) from node r to node c. The arrows represent the function *parent* and the dashed lines represent a correspondence function which is a bijection from the set of all objects being copied to the set of new objects (a', d', and e') which is a copy of that set. The correspondence bijection is used to maintain the structure of directory a in the copy.

Event Move $\widehat{=}$
Any
obj, to, des
Where
$grd1 : obj \in objects \setminus \{root\}$
$grd2 : to \in objects$
$grd3 : des \subseteq objects$
$grd4 : des = (tcl(parent))^{-1}[\{obj\}]$
$grd5 : to \notin des \cup \{obj\}$
Then
$act1 : parent(obj) := to$
End

Fig. 9. A specification of Event *Move*

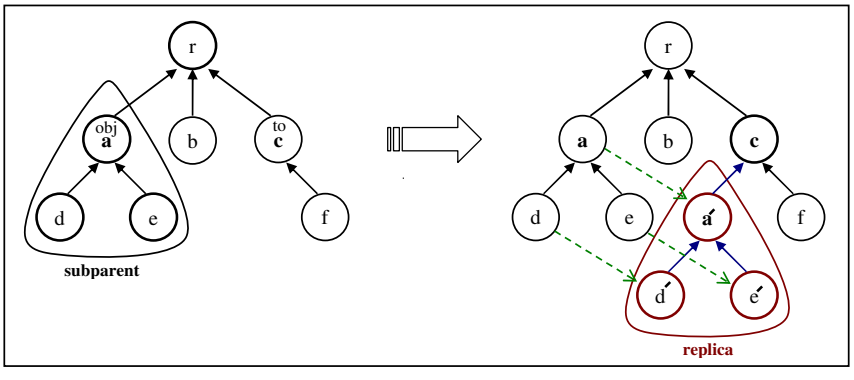


Fig. 10. A diagram of copying a subtree (*subparent*) rooted at a from r to c

Considering Event *Copy* given in Fig. 11, *obj* (the object being copied) and *to* (the destination) behave like external parameters provided by users or application programs, while the rest are local parameters used for computation. However, there is no distinction between external parameters and local parameters in Event-B. In this event, *des* is the set of all descendants of the object *obj* which is equal to $(tcl(parent))^{-1}[\{obj\}]$; *objs* is the set of all objects being copied; *nobjs* is the set of new objects corresponding to the set *objs*; *corres* is the correspondence bijection. With reference to Fig. 10, *subparent* represents the subtree rooted at a which is being copied. In this event, *subparent* is equal to $des \lhd parent$ which is a restriction of the parent function to *des* (e.g., $d \mapsto a$ and $e \mapsto a$ in Fig. 10). Finally, *replica* is a copy of *subparent* which is equal to $corres^{-1}; subparent; corres$ (e.g., $d' \mapsto a'$ and $e' \mapsto a'$ in Fig. 10).

Event Copy $\widehat{=}$
  Any
    obj, **to**, $des, objs, corres, nobjs, nobj, subparent, replica$
  Where
    $grd1 : obj \in objects \setminus \{root\}$
    $grd2 : des \subseteq objects$
    $grd3 : des = (tcl(parent))^{-1}[\{obj\}]$
    $grd4 : to \in objects$
    $grd5 : to \notin des \cup \{obj\}$
    $grd6 : objs = des \cup \{obj\}$
    $grd7 : nobjs \subseteq OBJECT \setminus objects$
    $grd8 : corres \in objs \rightarrowtail\!\!\!\!\!\!\rightarrow nobjs$
    $grd9 : nobj = corres(obj)$
    $grd10 : subparent = des \lhd parent$
    $grd11 : replica = corres^{-1}; subparent; corres$
  Then
    $act1 : parent := parent \cup replica \cup \{nobj \mapsto to\}$
    $act2 : objects := objects \cup nobjs$
  End

Fig. 11. A specification of Event *Copy*

Event Delete $\widehat{=}$
  Any
    $obj, des, objs$
  Where
    $grd1 : obj \in objects \setminus \{root\}$
    $grd2 : des \subseteq objects$
    $grd3 : des = (tcl(parent))^{-1}[\{obj\}]$
    $grd4 : objs = des \cup \{obj\}$
  Then
    $act1 : objects := objects \setminus objs$
    $act2 : parent := objs \lhd\!\!\!\!\!\!\rightarrow parent$
  End

Fig. 12. A specification of Event *Delete*

At this point, the reason we introduce a number of additional local parameters is to make models easier to read and prove. For example, without introducing *des*, *subparent* and *replica*, *act*1 can be replaced by

$$parent := parent \cup corres^{-1}; (tcl(parent))^{-1}[\{obj\}] \lhd parent; corres \cup \{nobj \mapsto to\}$$

but we can see that the action becomes more difficult to read.

Additionally, there are two main constraints in this event. Firstly, the object being copied, *obj*, must exist and must not be the *root*. This is satisfied by *grd*1. Secondly, the destination, *to*, must exist and must not be the object being copied or its descendant (satisfied by *grd*5). Guard *grd*5 plays an important role to ensure that loops are not produced by this event.

Delete event. This event is given in Fig. 12. In the figure, *obj* is an object being deleted; *des* is a set of all *obj*'s descendants. Here *grd*1 states that *obj* must be an existing object except *root*. The object being deleted and all its descendants, *objs*, will be removed from *objects* by *act*1 and all related parent-entries also be removed by *act*2.

6 First Refinement: Files and Directories

In this refinement, objects are classified as files or directories. There are two machine variables introduced in this level, namely, $files$ (a set of existing files) which is initialised to the empty set and *directories* (a set of existing directories) which is initialised to a set of *root*. The variables $files$ and *directories* are used to partition the variable *objects*. Additionally, the *Create* event of the abstraction is refined into events $crtfile$ (create file) and *mkdir* (make directory). Additional requirements for this level are given below.

Req2.1: Set of objects is partitioned into files and directories.
Req2.2: Root node is a directory.
Req2.3: The parent of each object must be a directory.

Variables
$files$, *directories*, *parent*
Invariants
$inv2.1 : files \subseteq objects$
$inv2.2 : directories \subseteq objects$
$inv2.3 : files \cap directories = \varnothing$
$inv2.4 : objects = files \cup directories$
$inv2.5 : root \in directories$
$inv2.6 : ran(parent) \subseteq directories$
Initialisation
$files := \varnothing$
$directories := \{root\}$
$parent := \varnothing$

Fig. 13. Machine variables, invariants and initialisation of the first refinement

Fig. 13 shows a list of machine variables, invariants formulated to satisfy above requirements and initialised values of each variable. Considering the gluing invariant $inv2.4$, the abstract variable $objects$ is entirely defined in terms of $files$ and $directories$. As a result, it can be substituted by $files \cup directories$ and is no longer used in this level.

In this refinement, we choose two events (*Create-file* and *Copy*) to illustrate a concrete model of this level.

Create-file event. This event (named $crtfile$), given in Fig. 14, refines $Create$ of the previous abstraction. Additional details introduced in this refinement: (i) $grd2$, in must be a directory; and (ii) $act1$, the object must be added to the set $files$ directly, instead of the set $objects$ in the previous abstraction.

Event crtfile refines Create $\widehat{=}$
  Any
    obj, in
  Where
    $grd1 : obj \in OBJECT \setminus (files \cup directories)$
    $grd2 : in \in directories$
  Then
    $act1 : files := files \cup \{obj\}$
    $act2 : parent(obj) := in$
  End

Fig. 14. A specification of Create-file event

A refinement of Event *Copy***.** In this refinement, see Fig. 15, additional details introduced in this event are: (i) $grd4$, the destination, to, must be a directory; (ii) $act2$, all correspondents of $objs$ which are files must be added to the set $files$; and (iii) $act3$, all correspondents of $objs$ which are directories must be added to the set $directories$ as well. These two actions refine Action $act2$ of the previous abstraction.

7 Second Refinement: File Content

In this refinement, file contents and other requirements related to the contents are introduced accompanied with four events: *open* (open an existing file), *read* (read the whole content of a file from the storage into memory buffer), *write* (write the content of a file on the buffer back to the storage) and *close* (close an opened file). Some constraints are covered in this level – such as each file has some content; each file must be opened before reading or writing; and a buffer of each opened file will be assigned once the file is opened and released when the file is closed. Machine variables introduced in this refinement are listed and discussed below.

Event Copy refines Copy $\widehat{=}$
Any
$obj, to, des, objs, corres, nobjs, nobj, subparent, replica$
Where
$grd1 : obj \in (files \cup directories) \setminus \{root\}$
$grd2 : des \subseteq (files \cup directories)$
$grd3 : des = (tcl(parent))^{-1}[\{obj\}]$
$grd4 : to \in directories$
$grd5 : to \notin des \cup \{obj\}$
$grd6 : objs = des \cup \{obj\}$
$grd7 : nobjs \subseteq OBJECT \setminus (files \cup directories)$
$grd8 : corres \in objs \rightarrowtail\!\!\!\!\twoheadrightarrow nobjs$
$grd9 : nobj = corres(obj)$
$grd10 : subparent = des \lhd parent$
$grd11 : replica = corres^{-1}; subparent; corres$
Then
$act1 : parent := parent \cup replica \cup \{nobj \mapsto to\}$
$act2 : files := files \cup corres[objs \cap files]$
$act3 : directories := directories \cup corres[objs \cap directories]$
End

Fig. 15. A refinement of Event *Copy*

$$fcontent \in files \rightarrow CONTENT$$

$$opened\_files \subseteq files$$

$$fbuffer \in opened\_files \rightarrow CONTENT$$

In this refinement, the content of files, $fcontent$, is defined as a total function mapped from each file to a content. Variable $opened\_files$ is set of files which are opened. The buffer of each opened file, $fbuffer$, is specified as a total function mapped from each opened file to a content. The content is an array of data items (BYTEs). In a context seen by this refined machine, the content is defined as a constant named $CONTENT$; and $BYTE$ is defined as a carrier set.

$$CONTENT = \mathbb{N} \twoheadrightarrow BYTE$$

Fig. 16 given below is an example of event *read*. This event is aimed at reading the whole content of a file named f from a storage into its buffer. Guard $grd1$ states that the file f must be an opened file.

8 Third Refinement: Permissions

In this level, requirements related to access permissions are introduced. For example, each object has an owner, a group-owner and a list of permissions. Access

Event read $\widehat{=}$
Any
f
Where
$grd1 : f \in opened\_files$
Then
$act1 : fbuffer(f) := fcontent(f)$
End

Fig. 16. A specification of Event *read*

to each object depends on its permissions. Additionally, users and groups are specified in this level as well. For instance, each user can be a member of one or more groups but at most one primary group is assigned, etc.

Considering Fig. 17, a number of machine variables are introduced in this refinement: *users*, a set of existing users; *groups*, a set of existing groups; *user_pgrp*, a primary group of each user; *user_grps*, user's groups; *obj_owner*, an owner of each object; and *obj_perms*, permissions of each object. Invariant *inv*4.5 states that a primary group of each user must be a group in which the user be a member. In a context seen by this machine, GROUP, USER and PERMISSION (a set of permission types) are defined as carrier sets.

Variables
...
$users$, $groups$, $user\_pgrp$, $user\_grps$, obj_owner, obj_grp, obj_perms
Invariants
$inv4.1 : users \subseteq USER$
$inv4.2 : groups \subseteq GROUP$
$inv4.3 : user\_pgrp \in users \rightarrow groups$
$inv4.4 : user\_grps \in users \leftrightarrow groups$
$inv4.5 : \forall u \cdot u \in users \Rightarrow user\_pgrp(u) \in user\_grps[\{u\}]$
$inv4.6 : obj\_owner \in (files \cup directories) \rightarrow users$
$inv4.7 : obj\_grp \in (files \cup directories) \rightarrow groups$
$inv4.8 : obj\_perms \in (files \cup directories) \leftrightarrow PERMISSION$

Fig. 17. Additional machine variables and invariants of the third refinement

Fig. 18 is an example of Event *read*, which refines the *read* event of the previous abstraction. In this event, guards *grd*2 and *grd*3 state that user *usr* who issues this read request must exist and has a read permission on f.

Event read refines read $\widehat{=}$
Any
f, usr
Where
$grd1 : f \in opened\_files$
$grd2 : usr \in users$
$grd3 : f \mapsto usr \in$
$RPerm(obj\_perms \mapsto obj\_owner \mapsto obj\_grp \mapsto user\_grps)$
Then
$act1 : fbuffer(f) := fcontent(f)$
End

Fig. 18. A refinement of Event *read*

RPerm, which is mentioned in the event *read*, encodes the rules that determine whether a user has read permission for a file. It is defined in a context seen by this machine. Part of this context which is related to *RPerm* is shown in Fig. 19. In the figure, p represents a permission relation; s is an owner function; g is an object-group function; m is a user-group relation; su, a super user (who has the right to manage every thing), is defined as a USER constant; and rbo (owner-read), rbg (group-read) and rbw (world-read) are permission types. This function states that a user u has a permission to read an object o only if at least one of these criteria is satisfied: (i) the user is the owner and has the owner-read permission; (ii) the user is a member of the group to which the object belongs and has the group-read permission; (iii) the world-read permission is assigned to the object; and (iv) the user is the super user. Other permission definitions (i.e., write and execute permission functions) which are not mentioned here are also specified in the same way.

$$
\begin{aligned}
& o \mapsto u \in RPerm(p \mapsto s \mapsto g \mapsto m) \\
& \quad \Leftrightarrow (\quad (o \mapsto u \in s \wedge o \mapsto rbo \in p) \\
& \qquad \vee \\
& \qquad (g(o) \in m[\{u\}] \wedge o \mapsto rbg \in p) \\
& \qquad \vee \\
& \qquad (o \mapsto rbw \in p) \\
& \qquad \vee \\
& \qquad (u = su) \quad)
\end{aligned}
$$

Fig. 19. A definition of read permission function

9 Proofs

The proof statistics, given in Fig. 20, show that 162 proof obligations were generated by the Rodin platform. 127 proof obligations (or 78%) were proved automatically while others were discharged by interactive proof. In the figure, MCH0 represents an abstract model; MCH1, MCH2 and MCH3 represent the first, second and third refinements of the abstract model. CTX0 up to CTX3 represent corresponding contexts which are seen by those machines.

Machines/Contexts	Total POs	Automatic	Interactive
CTX0	10	8	2
CTX1	7	3	4
CTX2	0	0	0
CTX3	3	3	0
MCH0	35	22	13
MCH1	50	42	8
MCH2	17	15	2
MCH3	40	34	6
Overall	162	127 (78%)	35 (22%)

Fig. 20. Proof statistics

In order to make proof easier and reduce the time required, we introduced proved theorems that could be reused for discharging some similar proof obligations. The example given in Fig. 21 is a theorem introduced in a context seen by the abstract machine. This theorem was used to prove that the no-loop property held for Event $Copy$. To prove this event preserves the no-loop property ($inv1.4$), we provided: $f = parent$, $g = replica$, $r = root$, $u = nobj$, $x = to$, $M = objects$ and $N = nobjs$. However, the theorem could be reused for Event $Create$ and $Move$ events as well. For example, in the case of Event $Create$, g was assigned to be the empty set, $u = obj$ and $N = \{obj\}$.

Proof of Theorem $thm4$ was one of the most complex of the interactive proofs. We outline the steps involved. Proving $thm4$, we have the goal $G1$:

$$C = \varnothing \tag{G1}$$

with the hypothesis $H1$:

$$C \subseteq (f \cup g \cup \{u \mapsto x\})^{-1}[C] \tag{H1}$$

$H1$ is rewritten to $H2$:

$$C \subseteq f^{-1}[C] \cup g^{-1}[C] \cup \{x \mapsto u\}[C] \tag{H2}$$

Now we cannot prove that $C \subseteq f^{-1}[C]$, but we can use $H2$ and other antecedents of $thm4$ to prove

$$C \cap M \subseteq f^{-1}[C \cap M] \tag{H3}$$

$$
\begin{array}{l}
thm4 : \forall f, g, r, u, x, M, N \cdot \\
\quad M \subseteq OBJECT \land\ N \subseteq OBJECT \land\ M \cap N = \varnothing \\
\quad \land\ r \in M \land f \in M \setminus \{r\} \to M \\
\quad \land\ u \in N \land g \in N \setminus \{u\} \to N \\
\quad \land\ x \in M \\
\quad \land\ (\forall A \cdot A \subseteq f^{-1}[A] \Rightarrow A = \varnothing) \\
\quad \land\ (\forall B \cdot B \subseteq g^{-1}[B] \Rightarrow B = \varnothing) \\
\quad \land\ f \cup g \cup \{u \mapsto x\} \in (M \cup N) \setminus \{r\} \to M \cup N \\
\Rightarrow \\
\quad (\forall C \cdot C \subseteq (f \cup g \cup \{u \mapsto x\})^{-1}[C] \Rightarrow C = \varnothing)
\end{array}
$$

Fig. 21. A theorem used for discharging no-loop proof obligation

From $H3$ and the antecedent of $thm4$ we can prove

$$C \cap M = \varnothing \tag{H4}$$

Similarly, we can prove that

$$C \cap N = \varnothing \tag{H5}$$

We observe that C can be partitioned by M and N. Thus, using $H2$ we can prove

$$C = (C \cap M) \cup (C \cap N) \tag{H6}$$

Finally, $G1$ is proved using $H4$, $H5$, and $H6$.

10 Comparison with Related Work

A number of formalisations of file systems have been developed by other researchers. Most of them are focused on file contents, and read and write operations. There is some work that deals with the structure of file systems. A specification of a visual file system in Z by Hughes [12] is focused on a tree structure and operations affecting the tree structure, but file content and a manipulation of file content were not specified. In this specification, transitive closure was chosen to specify main property of a tree structure, e.g. reachability. However, the no-loop property was not mentioned in this specification. In addition, this specification had no refinement and no proof. Another related work by Morgan and Sufrin presented in [11] is a specification of a Unix filing system in Z. In this specification, instead of using a tree structure, the location of each object is formulated as a sequence of directory names, which is the path of each file. This work is concentrated on file contents and naming operations used for manipulating these rather than structure manipulation operations such as directory copy and move. Based on the specification of Morgan and Sufrin, Freitas,

Fu and Woodcook [10] have developed a verified model of the POSIX filestore accompanied with a representation and proof using the Z/Eves proof system.

Since the filestore challenge was proposed by Joshi and Holzmann [13] in 2005, other researchers have addressed this challenge. For example, Butterfield and Woodcook [7] have developed an abstract specification in Z of the ONFi specification [16]. In addition, Ferreira et al. [9] have developed and verified a specification of the Intel Flash File System Core [17] in VDM. Alloy and HOL were used as tools for model checking and theorem proving. Another work contributed by Kung and Jackson [14] is a formal specification and analysis of a flash-based file system in Alloy. This work focused on basic operations of a filesystem and features covering wear-levelling and fault tolerance.

11 Lessons and Conclusion

In this paper, we have presented a verified model of a tree-structured file system focusing on the tree structure and the basic operations affecting the tree structure. Our aims are constructing a model with clear and accurate formulation of the system properties and discharge of all proof obligations. To satisfy these, careful selection of invariants and machine theorems was important and eased the proof effort. For example, for the high-level requirements on the data structure, we introduced two tree properties: (i) no-loop and (ii) reachability. Both these properties are naturally expressed using transitive closure. However, we identified simpler but sufficient formulations ($inv1.3$, $inv1.4$) and exposed these as invariants. Proving that all events preserved these invariants was not too difficult since they did not involve transitive closure. The transitive closure formulations were expressed as machine theorems and we proved that these followed from the machine invariants. We did not need to prove that the theorems are preserved by all machine events which simplified the proof effort considerably.

Our experience of using the Rodin tool was very positive[1]. The supported language was sufficiently expressive and all proof obligations could be discharged. We achieved a good degree of automatic proof. All interactive proofs involved a small number of steps and were straightforward to achieve.

Based on this experience, we have found that general theorems should be specified in a context such as Theorem $thm4$ in Fig. 21. They can be seen and used by more than one machine, and can be extended by other contexts. Specific theorems which are derived from machine variables and invariants should be specified in machines (such as machine theorems given in Fig. 6). These machine theorems can be used to help discharge proof obligations as well. Introducing additional theorems that can be reused for discharging similar proof obligations makes automatic and interactive proof easier and can reduce the time required for proofs. In addition, instead of introducing new machine invariants to satisfy system properties, providing machine theorems and proving that those properties

[1] Caveat: Abrial and Butler are developers of the Rodin tool so are not objective evaluators.

are satisfied is another mechanism used to specify system models. This mechanism can reduce the number of proof obligations and makes models simpler and easier to prove.

Additionally, it can be seen in an example given in Section 5, providing additional parameters in each event is useful sometimes. Although more guards are needed, it could make models more readable and easier to manage in both specifying and proof.

Refinement can be used to introduce other requirements that may be postponed or missed from the previous steps and later be covered in the refinement steps. Refinement allows us to factor out some of the modelling and proof complexities. In this development we chose to focus on the tree structure manipulation in the abstract model and postpone other details to later refinements - for example, we do not distinguish files and directories at the abstract level. This made the proof obligations and invariants for the tree structure easier to formulate than if we had tried to model everything in one level. Note that we regards the full chain of refinements as constituting *the specification*, not just the most abstract level.

Finally, it can be stated that this example allowed us to define a kind of modelling methodology – finding the right mathematical concepts, finding useful general theorems – which could be exported in many different complex modelling projects which require a manipulation of the tree structure.

References

1. Abrial, J.-R.: The B Book. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R.: A system development process with Event-B and the Rodin platform. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 1–3. Springer, Heidelberg (2007)
3. Abrial, J.-R.: Modelling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2008)
4. Abrial, J.-R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260. Springer, Heidelberg (2006)
5. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamentae Infomatica, 1001–1026 (2006)
6. Butler, M.: Rodin deliverable D31: Plublic versions of plug-in tools. Technical report, University of Southampton, UK (2007)
7. Butterfield, A., Woodcock, J.: Formalising flash memory: First steps. In: 12th ICECCS 2007, pp. 251–260. IEEE Computer Society Press, USA (2007)
8. Coleman, J., Jones, C., Oliver, I., Romanovsky, A., Troubitsyna, E.: RODIN (Rigorous open Development Environment for Complex Systems). In: 5th European Dependable Computing Conference: EDCC-5 supplementary, Budapest, pp. 23–26 (2005)
9. Ferreira, M.A., Silva, S.S., Oliveira, J.N.: Verifying Intel flash file system core specification. Technical report, University of Minho (2008)
10. Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: an experiment in the verified software repository. In: 12th ICECCS, pp. 3-14 (2007)

11. Hayes, I.: Specification Case Studies. Prentice Hall International, UK (1992)
12. Hughes, J.: Specifying a visual file system in Z. Technical report, Department of Computing Science, University of Glasgow (1989)
13. Joshi, R., Holzmann, G.J.: A mini challenge: Build a verifiable filesystem. In: Verified Software: Theories, Tools, Experiments (2005)
14. Kang, E., Jackson, D.: Formal modeling and analysis of a flash filesystem in Alloy. 1st Conference on ASM, B, and Z (ABZ 2008). London, UK (to appear) (September 2008)
15. Métayer, C., Abrial, J.-R., Voisin, L.: Rodin deliverable 3.2. Event-B language. Technical report, University of Newcastle upon Tyne, UK (2005)
16. Cemicondutor, H., et al.: Open NAND Flash Interface Specification. Technical report Revision 1.0, ONFI (December 2006), `http://www.onfi.org`
17. Intel Flash File System Core Reference Guide, version 1. Technical report 304436001, Intel Coorporation (October 2004)

Conformance Testing Based on UML State Machines

Automated Test Case Generation, Execution and Evaluation

Dirk Seifert

LORIA — Université Nancy 2, Campus scientifique, BP 239
54506 Vandœuvre lès Nancy cedex, France
Dirk.Seifert@Loria.fr

Abstract. We describe a comprehensive approach for conformance testing of reactive systems. Based on a formal specification, namely UML state machines, we automatically generate test cases and use them to test the input-output conformance of a system under test. The test cases include not only the stimuli to trigger the system under test, they also include the test oracles to automatically evaluate the test execution. In contrast to Harel Statecharts, state machines behave asynchronously, which makes automatic test case generation a particular challenge. As a prerequisite we have completely formalized a substantial subset of UML state machines that includes complex structured data. The TEAGER tool suite implements our test approach and proves its applicability.

1 Introduction and Related Work

The impact of embedded systems in our everyday life is steadily growing. They are present not only in very specific contexts, but also in nearly every electrical device we use. In general, embedded systems comprise of hardware and software components interacting with a specialized technical environment via sensors and actors. The main reason for their success is the combination of specific or high-performance hardware with the flexibility of software. The software is responsible for controlling the hardware and software components and for calculating reactions as responses to received events. Erroneous systems annoy the costumers and are a high commercial risk in mass customization. Moreover, size and complexity of nowadays systems demand for improved and automated processes: for development as well as for quality assurance.

In a model-based development approach, models of the system which have to be built, guide and control the development process. There are various types of models differing in the level of abstractions or in their intended use. In the first steps, the models are used to analyze the problem domain and to ease the information exchange among developers. Later on, they form the basis to design and implement the system, serve as documentation, and are also used for quality assurance purposes. For example, the Unified Modeling Language (UML) comprises thirteen diagram types to specify the structure and the behavior of a system or a system component [1]. The included state machines are used to either describe the discrete reactive behavior (behavioral state machines) or to describe the usage protocol (protocol state machines). In our approach we use the behavioral state machines to specify the states a system can take and actions

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 45–65, 2008.

it can execute during its lifetime in response to internal and external events. The discrete reactive character of state machines and the possibility to completely specify the behavior of a system make state machines appropriate to model reactive systems. They also allow to automatically generate test cases that include test oracles. Testing means executing a system under test (SUT) with selected but real data to evaluate its conformance, whereat conformance is evaluated on the basis of the observations made on the system under test. It aims in falsification, that means to show inconsistencies between the specification and the SUT. It benefits from the fact that the actual system is brought to execution, and thus, the interaction of the real hardware and software can be evaluated. It also benefits from the fact that it is applicable at different levels of abstraction and at different stages of the development.

The contribution of our work is twofold: first, we formalized a substantial subset of UML state machines that includes the manipulation of complex structured data. Second, we present a conformance test approach based on UML state machines that allows automatic generation, execution and evaluation of test cases on the level of unit testing. To our knowledge there is no formalization related to the latest UML standard [1] that precisely formalizes data aspects. We focus on the reactive behavior and skip real time aspects in this paper (in short: we allow to manually specify lower and upper timeout limits for every event). Considering real time behavior is still a challenge for automated processes. It is one of our prospects for future work.

Our formalization of UML state machines is influenced by several work (e.g., [2–4]). Most approaches focus on model checking and the chosen representation is not always appropriate for automated test case generation. Moreover, there have been major changes in UML 2 that require a revision of previous work. We present an *operational* semantics that is complete with respect to the considered subset and includes all necessary definitions. Moreover, it is the first formalization which includes all definitions related to complex structured data. De Nicola and Hennessy [5] introduce a formal theory of testing on which (later on) Brinksma [6] and Tretmans [7] build approaches to derive test cases. For basic work on testing based on transition systems and (extended) finite state machines we refer to the surveys [8–10]. A lot proposals only deal with deterministic systems and require that the model must be strongly connected [11–13], or assume the testing process to communicate synchronously with the system under test [6, 10]. More recent research allow some of these requirements to be relaxed (see e.g. [14–16]) but most refer to older UML standards, consider different subsets, or do not consider complex structured data. Moreover, we allow nondeterminism in the specification as well as in the SUT, and asynchronous communication between the SUT and its environment. A further difficulty in using transition or finite state machines based techniques is that transitions in state machines are labeled with *input-output pairs* associated to a *single* transition. The underlying semantic model builds on the *steps* of the whole state machine. Those comprise the execution of several transitions including changing the current configuration and executing actions such as those that generate output events. In such a step the atomicity of input processing and output generation is preserved. Mapping the semantics to transition systems or input/output automata to apply classical techniques would require to introduce intermediate states and transitions. On the one hand this would break the correspondence of a state machine step

and a semantic step, and on the other hand such a multi-step approach unnecessarily complicates the formalization and use of state machines.

Latella et al. [15, 17, 18] follow (as we do) a "semantics-first" approach in which a sound basic kernel of the notation is considered and extended, only if the main features are investigated. In contrast to our work they do not consider complex structured data (i.e., interpret a state machines data space or event parameters). They also refer to an older UML standard and do not consider different transition orders, which become relevant if data are regarded. Offutt et al. [19, 16] present techniques to generate test cases from UML state diagrams on class-level testing. In contrast to our work, they have a data-centric view and focus on change events and boolean variables. The generated test suites are related to full-predicate coverage. Moreover, it is not clear how far the applied semantics follows the UML standard. The work around the AutoFocus tool [20] is interesting but they use proprietary notation which does not include all aspects of UML state machines and a formal, but synchronous, semantics. Another mentionable industrial approach are the AGEDIS tools [21], but the used semantics is not completely clear. Spec Explorer developed at Microsoft Research ([22] and related publications) is an industrial approach that uses finite state machines as the underlying model for automated testing. They also test against nondeterministic systems and address problems of data instantiation. The general principle to explore the specifications state space is comparable to our approach. In contrast, the focus is on (synchronous) method calls.

In Sec. 2 we introduce the syntax and semantics of state machines we need in this paper by means of an example. In Sec. 3 we present our test approach to automatically generate test cases out of a state machine specification. We describe the underlying theory, the test case generation algorithms, how approximation techniques are used to increase the efficiency and how the test case generation and execution can be controlled and evaluated. In Sec. 4 we present our TEAGER tool suite and discuss experimental results. In Sec. 5 we conclude our work and give an outlook to ongoing research.

2 State Machines

UML state machines [1] are an object-oriented extension of the classical Harel Statecharts [23]. We use them to describe the sequence of states a system or system component can take and the actions it executes when changing these states. State machines are mathematical models with a graphical representation: the nodes depict simple or composed states of the system and the labeled edges depict transitions between these states. Composite states are used to hierarchically and orthogonally structure the model, thus reducing its graphical complexity. Labels express conditions under which transitions can be taken and the actions that will be executed when the transitions are taken. Events are used as triggers to activate transitions and can be parameterized to exchange data. Optional, every state machine has a data space that can be read and manipulated by the state machine during execution. More precisely, it is possible to read data values to describe specific conditions when a transition can be taken or to manipulate data values and exchange information within actions. A transition comprises a source state, a trigger event, an optional guard, an optional effect (which consists of a sequence of actions), and a target state. A guard describes a fine-grained condition (with reference to the system's state) that must evaluate to true to enable the transition. Hence, the activation

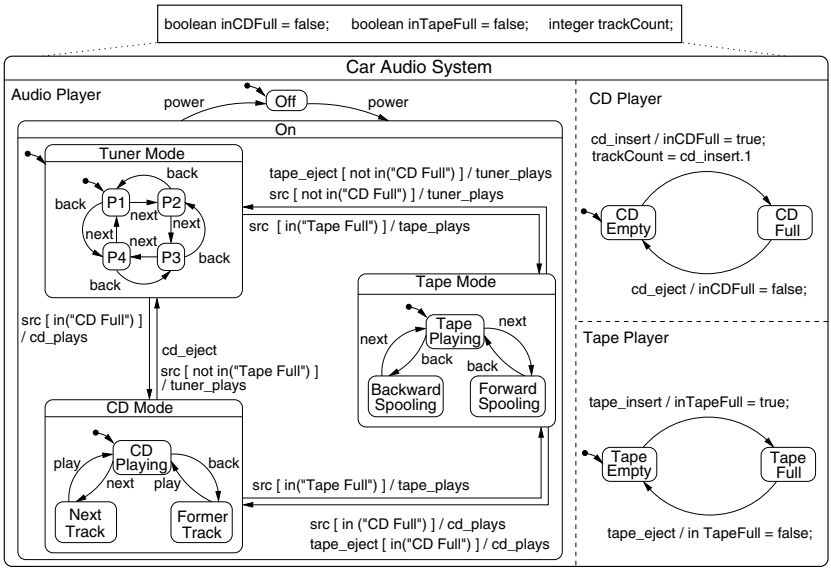


Fig. 1. State machine specification for the Car Audio System

of the source state, the trigger event and the fulfilled guard condition constitute the necessary constraint to fire a transition. An action can either be a statement manipulating the data space or the generation of new events. The action sequence and the subsequently active target state constitute the overall effect of the transition. In opposite to the classical Statecharts, the event processing takes place in a so-called *run-to-completion* step [1]. This asynchronous event processing demands the processing of the previous event to be completely finished before the next event can be processed. In the following we briefly describe state machines by means of an example. Afterwards we discuss semantic issues which pose a challenge for automated test case generation. A complete and detailed description as well as a precise definition of the semantics (including the integration of complex data) can be found in [24, 25].

2.1 Example

We use a state machine model specifying the behavior of a simple sound device in a car to demonstrate the state machine notation. The requirements for such sound device could be as follows: *It should be possible to turn the Car Audio System on and off. When turned on, it should play one of three different audio sources, namely radio, tape or compact disc, respecting the presence of a tape or a compact disc. It should be possible to change between available sources. Furthermore, it should be possible to switch between four radio stations, to spool a tape backward or forward, or to select the previous or the next track of a compact disc.*

We introduce the following events to model the required behavior: `power`, `src` (to switch between the different sources), `next`, `back` and `play`. We also introduce events signaling the insertion and the ejection of a tape or a compact disc as well

as events to signal system reactions. Furthermore, we use data variables to store detailed information about the current state. For example, we use an integer variable `trackCount` to store the number of titles of an inserted compact disc. Figure 1 shows a state machine model of the sound device including the related data space.

At the highest level of abstraction the model consists of an orthogonal state comprising three regions. The two regions `CD Player` and `Tape Player` model the information if a tape or a compact disc is inserted into the system or not. The more complex region `Audio Player` models the control of the system. The region is refined by two states: `Off` and `On`. Initially the system is assumed to be switched off, expressed by the small arrow leaving a bullet and ending at the `Off` state. When the event *power* is processed the system is switched on and starts to play the radio (again expressed by a small arrow). The composite state `On` is refined into states modeling the three signal sources. The transitions between these states describe the changes between the sources as reaction to an event *src*. For example, when the system is in `Tuner Mode` and a tape and a compact disc are inserted into the system (i. e. both in-predicates are true) and the event *src* is processed, the system can either switch to the tape mode or switch to the compact disc mode because both transitions are enabled and can fire. All three substates of `Audio Player` are further refined to describe the particular behavior in reaction to the events `next`, `back` and `play` in each state.

2.2 State Machine Semantics

The semantics of UML state machines is adapted from the STATEMATE semantics [26] to fit into the object-oriented paradigm. As described above a state machine can be refined by simple composite and orthogonal states. Simple composite states contain exactly one region and orthogonal states contain at least two regions. In every region only one substate can be active at a time. The state which is entered by default when a region is entered is marked by an arrow emanating from a filled circle. The hierarchical ordering of states forms a tree structure with a region as the root node, simple states at the leave nodes and in between (alternating) composite states and regions.

Due to orthogonal regions a state machine can have several active states at a time. We call the set of all active states *configuration*. For the same reason it is possible that more than one transition can fire at a time — one in every active orthogonal region. We call the set of all jointly firing transitions *firing transition set* (FTS). Due to the hierarchical structure it can happen that two transitions are enabled for firing on different hierarchy levels of a state. Taking both would lead to a configuration which is not well-formed. A similar situation arises if a transition leaves an orthogonal region. In this case the transition cannot fire together with an enabled transition in another orthogonal region. In both cases the transitions are said to be in conflict with each other. Conflicts are identified if two transitions leave identical states in the state hierarchy. The UML describes a two-step process to resolve conflicts. In the first step a priority scheme is used. A transitions emanating from a state deeper in the state hierarchy has priority over the other transition. Thus the more refined transition is taken. This differs from classical Statecharts, but it reflects the object-oriented inheritance behavior. However, not all conflicts can be resolved using this priority scheme. In the second step only transitions are selected which are not in conflict to each other allowing for maximal progress of

the system. A so-called *transition selection algorithm* selects all maximal sets $T_{\|} \subseteq T$ of enabled transitions fulfilling the following requirements:

$$\forall t : T_{\|} \bullet enabled(t,c,e,d) \tag{1}$$

$$\forall t_1, t_2 : T_{\|} \mid t_1 \neq t_2 \bullet t_1 \parallel t_2 \tag{2}$$

$$\nexists\, t' : T \setminus T_{\|} \mid enabled(t',c,e,d) \bullet \forall t : T_{\|} \bullet t \parallel t' \vee t' \prec t \tag{3}$$

First, all transition in the set must be enabled regarding the current configuration, the trigger event and the current data assignments. Second, all transitions in the set are mutually conflict free (expressed by the $\|$ operator). Third, there is no enabled transition outside the set which is conflict free with all transitions in the set or with higher priority than a transition inside the set. Thus, transitions with the highest priority are taken and maximal sets are chosen. Result is a set of firing transition sets (FTSs). It is important to mention that for execution one FTS is arbitrarily chosen, and that the order in which the transitions in this set are fired is arbitrarily chosen, too. In consequence, all set choices and transition permutations form the set of all possible semantic steps of the state machine at a time. This is important if we want to compute the possible correct behavior for an input sequence to evaluate the test execution. In opposite to classical Statecharts, the event processing takes place in a so-called *run-to-completion* step. This asynchronous event processing demands the processing of the previous event to be completely finished before the next event can be processed. Therefore it is necessary to buffer received events in an event store. Consequently, the occurrence of an event and its processing are asynchronous (i.e., take place at different times). It follows that a possible (observable) reaction of the system also takes place asynchronously.

The semantic model of state machines builds on the semantic steps a state machine can execute during its lifetime. Such a step moves the state machine from one semantic state to another semantic state while receiving events from and emitting events to the environment. A semantic state (called a *status*) comprises three components, namely a configuration (a set of active states), an event store, and the variable assignments. We depict the components of a status in double square brackets $[\![c,q,d]\!]$ and a semantic step $[\![c,q,d]\!] \xrightarrow{in,out} [\![c',q',d']\!]$. Note that the chosen set of firing transitions and the execution order of these transitions can be identified (if necessary) from this representation. Assuming a state machine to be input enabled (cf. the next section) a *semantic step* can be described as follows:

$$\frac{q = <>}{[\![c,q,d]\!] \xrightarrow{E_{in},\langle\rangle} [\![c,q',d]\!]} \tag{4}$$

$$\frac{\begin{array}{c} q \in \mathrm{ran} \oplus \\ (q'',e) = \ominus(q) \\ c' = (c \setminus \bigcup_{\forall t:T_{\|}} exits(t)) \cup \bigcup_{\forall t:T_{\|}} enters(t) \\ A_{seq} \in perm(\{t : T_{\|} \bullet effect(label(t)(e))\}) \\ (d', E_{gen}) = performAll(\frown / A_{seq})(d) \\ (E_{int} = E_{gen} \upharpoonright E_{SM}) \wedge (E_{out} = E_{gen} \upharpoonright E_{env}) \\ q' = (q'' \oplus E_{int}) \oplus E_{in} \end{array}}{[\![c,q,d]\!] \xrightarrow{E_{in},E_{out}} [\![c',q',d']\!]} \tag{5}$$

We have to distinguish two cases. The first case covers the situation when the event store does not contain any event (4). During the step, only the events received from the environment (E_{in}) are added to the event store ($\oplus(q, E_{in})$). The active configuration and the data assignments are left unchanged. The second case covers the situation when the event store contains events for processing (5). During the step, the trigger event will be selected from the event store ($\ominus(q)$). The next configuration c' results from leaving all states the transitions exit, and entering all states the transitions enter. Next, an execution order for the FTS is chosen (*perm*), and the effect of this transition sequence is calculated (*performAll*). The effect includes the new data assignments (d') and the sequence of newly generated events (E_{gen}). Finally, this event sequence is processed. The generated internal events (E_{int}) and the events received from the environment (E_{in}) are added to the event store. The remaining external events (E_{out}) are sent to the environment. Now we can describe the execution of a state machine based on this definitions as a concatenation of semantic steps. We call such a sequence of semantic steps a *computation*.

$$[\![c_1, q_1, d_1]\!] \xrightarrow{in_1, out_1} [\![c_2, q_2, d_2]\!] \xrightarrow{in_2, out_2} \ldots \xrightarrow{in_{n-1}, out_{n-1}} [\![c_n, q_n, d_n]\!]$$

We use this execution model to define our test approach. Only a precise and clearly interpretable mathematical model as we presented here offers the basis for automated processes. Our complete state machine semantics can be found in [24, 25].

3 Test Case Generation

In the UML standard and in our semantics, too, not all semantic details are completely determined. These open issues are called *semantic variation points*. They prevent unnecessary restrictions in the semantics and allow some degrees of freedom for the implementation of the semantics. The user has to instantiate them before working with the semantics. Unfortunately, many problems with the UML semantics arise from semantic variation points. On the one hand some of them are not obvious in the standard and on the other hand decisions taken are often not propagated to the public. Concerning our test approach the most interesting semantics variation points are: the nature of the event store, events not enabling any transition, the selection policy of possible firing transition sets, and the execution order of the transitions in a chosen set.

We instantiate the two first semantic variation points and do not instantiate the latter two, thus the test approach works correctly for different implementations of a state machine specification. Precisely, we neither want to restrict how to choose a possible set of firing transitions (if there is more than one) nor do we want to restrict the order these transitions will be executed. This is different for the event store. In order to be able to calculate the possible correct behavior allowed by the state machine specification, we need to know the nature of the event store, or with other words, we have to decide for a specific nature. In most practical contexts a FIFO queue is used to store events for further processing. Hence we assume an unbounded reliable FIFO queue as event store. Second, we assume that events that do not enable a transition when they are processed are deleted and the next event from the event store will be processed. This implies that the state machines do not block. Technically they are called *input enabled*. In summary,

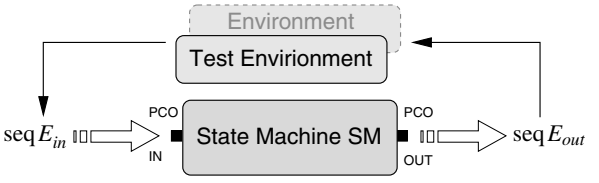


Fig. 2. Abstract test assembly for reactive systems

an event queue is introduced into the semantic model of state machines, non-enabling events from the queue will be omitted, and we need to respect different firing transition set selection and execution strategies in our test approach.

3.1 Conformance Relation for State Machines

As mentioned in the introduction, an embedded system usually comprises of hardware and software components. Hence, we treat the SUT as a black box to reflect this circumstance. We only require the SUT to have so-called *points of control and observations*. With these it is possible to control and observe the SUT. (i.e., to send inputs and to observe the outputs of the SUT). Figure 2 shows this abstract test assembly. As a consequence only the inputs to the SUT and the outputs of the SUT are visible in the environment and thus for the tester. This particularly implies that the event queue is not visible from the outside. To generate test cases and especially the test oracles we need to restrict the test generation to the observable parts of a SUT, but must respect internal details, which influence the possible behavior. Consequently, we need to extract the observable parts of the computations we defined for the semantic model of state machines. These are the events received from the environment and the generated events sent to the environment. Corresponding to the computation defined above we yield an *observable computation* by extracting and concatenating these events:

$$in_1 \frown out_1 \frown \ldots \frown in_{n-1} \frown out_{n-1} \tag{6}$$

We assumed the event store to be a queue so that received events will be stored one after another in sequence. Furthermore, we assume that transitions and actions on transitions are executed in sequence. Hence, generated events are also stored in sequence. The set of all observable computations form our observable execution model of state machines and the basis for the test case generation.

A prerequisite to automatically evaluate whether a SUT conforms to its specification is a precise definition of conformance. De Nicola and Hennessy studied various possible characterizations of conformance [5, 27]. Brinksma and Tretmans studied various implementation relations for synchronous transition systems [6, 7]. In general, relevant implementation relations are based on the same idea of an external observer. Here, an implementation I conforms to its specification S, if and only if, all observations obs any external observer $o : \mathcal{O}$ can make on the implementation, can be related to the observations this observer can make on the specification:

$$I \leq_o S \Leftrightarrow \forall o : \mathcal{O} \bullet obs(I,o) \sqsubseteq obs(S,o) \tag{7}$$

To get an applicable relation you need to define the type of observers ($\mathcal{O}$), which observations these observers can make (*obs*), and how to relate these observations ($\sqsubseteq$). We use sequences of inputs to the SUT as observers. The observations these observers can make are the resulting outputs (i.e., the generated events). The relation we use to compare observations of the system under test with the observations of the specification is set inclusion ($\subseteq$). We argue that a system under test conforms to its specification, if and only if, the output sequences for all possible input sequences are included in the set of all output sequences of the specification for the same input sequence (8). Following Tretmans [7] we restrict the set of possible inputs to that of the specification ($\mathrm{seq}\, E_S$). The set of outputs we calculate from the set of observable computations of a specification (9). Precisely, the set of all observations $out(S,\sigma)$ for S with input sequence σ results from all observable computations of S ($otraces(S)$) for which σ denotes the input sequence ($\sigma = \delta \upharpoonright E_S$) and $\delta \upharpoonright E_{env}$ denotes the resulting output sequence.

$$I \leq_{out} S \Leftrightarrow \forall \sigma : \mathrm{seq}\, E_S \bullet out(I,\sigma) \subseteq out(S,\sigma) \tag{8}$$

$$out(S,\sigma) == \{\delta : otraces(S) \mid \sigma = \delta \upharpoonright E_{in} \bullet \delta \upharpoonright E_{out}\} \tag{9}$$

Now we have a precise meaning of conformance and a guideline how to compute test cases and test oracles: based on the specification we need to calculate the traces of the state machine for all possible inputs and extract the possible correct observations. For testing we need to stimulate the SUT with the particular inputs, observe the outputs and compare them to the pre-calculated possible correct observations. That means to check for their existence. Obviously a problem arrises when thinking about practical testing — the set of inputs is usually infinitely large or at least pretty huge.

3.2 Selecting Inputs for Test Case Generation

When testing in practice we are only interested in relevant and interesting test cases to advantage the quality assurance process, and to use time and computation power at an optimum. Therefore, we generate test cases for pre-selected input sequence. This two-step process clearly separates the input selection problem from the test case generation problem. Hence it is possible to use different selection strategies with the same generation process and it allows to adapt the input selection process to different test purposes or to different project stages.

In the TEAGER tool suite we implemented several input selection strategies. The strategies range from using given fixed input sequences to using specific models describing the environment. The former allows special value testing and is used for very specific test purposes like the coverage of a certain path or state. The latter allows to model varied behavior of an environment based on probabilities. The most general one is an environment in which all inputs can happen at any time with the same probability (*uniform distribution*). In a more specific environment different probabilities are assigned to the inputs (*a prior distribution*). Thus the occurrence of specific inputs can be influenced. We also use a variant of this strategy where we adapt the probabilities once an input is chosen (*dependant distribution*). For every input a weight is assigned and decremented if the input is selected. If all weights are equal to zero the initial assignments will be used. With this strategy we ensure that eventually every event is chosen.

The most expressive way to describe the environment is to model it with probabilistic state machines. Using state machines allows to model dependencies among inputs in a sequence. It also allows to completely reassign input probabilities depending on the assumed state of the system under test. For example, the probability of dialing a number before lifting the receiver of a telephone is certainly different from the probability of dialing after lifting the receiver. In summary, we use different complex strategies to describe assumed environments to select relevant and interesting inputs. For a detailed description of our input selection strategies we refer to [28].

3.3 Test Case Generation Algorithm

During test case generation we consider a finite set of finite sequences of inputs and calculate all possible correct observations for these inputs. We use these observations to automatically evaluate the test execution process. Considering complex data during the test case generation process is not scope of this paper and we skip the corresponding details here. The problem which specific data to choose is part of ongoing research. In the current approach data are chosen randomly during the test case generation.

To calculate the possible correct observations we stepwise explore the state machine's state space for the given input. The challenge is to correctly consider all semantic subtleties. We do this in a two step algorithm. *First*, we initialize the state machine with its initial status (i.e., with its initial configuration, an empty queue and an initial data assignment). We insert the first input event to the event queue and apply a semantic step to this configuration. This includes that we calculate all possible FTSs. For every FTS and every possible execution order of the transitions inside these sets we calculate the resulting status. It is important to note that we calculate a fix-point for this set. That means, that no new status can be reached from any calculated status. Thus we yield a set of all reachable status including all intermediate status for the first event. To store the intermediate status is important for handling possible interleavings of input and internally generated events. *Second*, we insert the next event to every reachable status in the previously calculated set. Thus we respect possible interleavings of events in the event queue. Then we again calculate all reachable status for this input and proceed in the same way for the remaining inputs. Consequently, we calculate the graph of all execution paths including the reachable status. Only this stepwise calculation of all reachable status ensures that all possible execution paths for the given input sequence are calculated. This includes all non-determinism in the specification (modeled and arising from the semantic model of state machines) and effects from processing events asynchronously. Figure 3 visualize the principle of the calculation of an execution graph for the abstract input sequence `[a,b,c]`. The dashed parts show the newly calculated parts in the subsequent step. For example we can see that by processing the queue `[a,b]` we reach different status than by processing both events separately.

In the following we illustrate the key point of the algorithm by means of an short abstract example. Let us assume that an internal event `i` is generated when processing an event `a`. Let us further assume that processing an `i` will produce an internal event `j`. For the next test case we want to process the input sequence `a`·`b`. During test execution we have to trigger the SUT first with `a` and then with `b` with an (currently) undefined time gap between the events. The challenge is that we cannot predict the actual queue

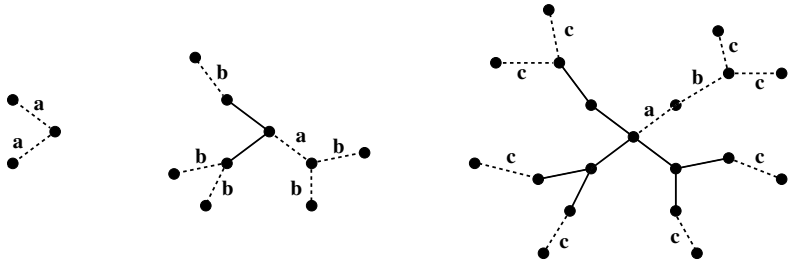


Fig. 3. Stepwise state space exploration for the input sequence `[a,b,c]`

of the SUT during test case generation. Consequently we do not know how event `b` will interleave with the internally generated events `i` and `j`. For this reason we first insert `a` into the queue and calculate the three reachable status: `[i],[j],[]`. The first queue results from just processing `a`. The second results from processing `a` and then `i` and the third results from processing `a`, then `i` and then `j`. By inserting `b` into all reached queues we prepare for respecting all possible interleavings. The resulting queues are `[i,b], [j,b], [b]` and during the next step `[b,j]` which properly respects one possible interleaving. Event `b` will also be inserted to the queue `[a]` resulting in queue `[a,b]`. This reflects the situation that we triggered both inputs before the system under test processed the first one. Figure 3 visualize this situation in the second graph.

After processing all events from the input sequence we can identify among the set of all reached status those status which are finally reached. These status (located at the hull of the execution graph) are *quiescent*. That means that their event queue is empty and thus they cannot proceed without a new input from the environment. Figure 4 shows at the right side an execution graph with the reached status at the hull. We extract from these the observations that would be emitted (i.e., the events the state machine sends to the environment) when executing this particular path. The extracted observation sequences comprise all possible correct observations we can make when triggering the system under test with the input sequence. Now the idea is to treat all observations as the alphabet of a language and the calculated observation sequences as accepted words of these language. Accepted observation sequences cause the test execution to pass. All other sequences cause the test execution to fail. Now we just need to build an acceptor for the calculated observation sequence and use them as the test oracle.

Finally, we need to overcome one open problem which can arise when calculating the execution graph. We previously mentioned that we determine for an input sequence the fix-point for the reachable status. Due to the fact that state machines can generate (internal) events and produce internal infinite loops the calculation of these fix-points does not terminate in any case (we also subsume the problem that the time to calculate the fix-point is unacceptable high). To overcome this problem we limit the number of steps needed to calculate all subsequent status to an upper bound. Technically, every reached status has got a counter for the number of steps necessary to reach this status. If a counter reaches the specified upper bound we mark this status and abort further processing of this status. Figure 4 shows this in the lower left corner.

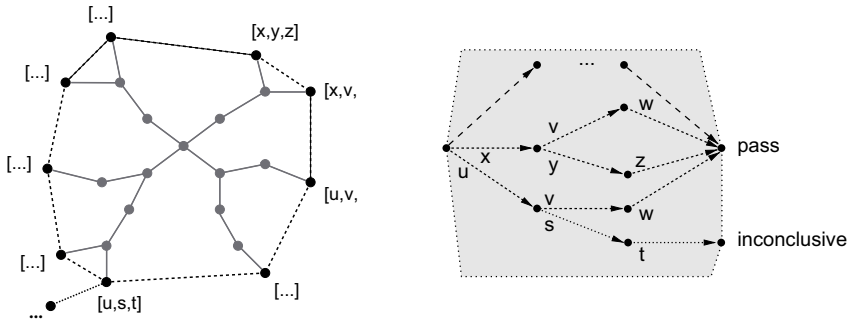


Fig. 4. Execution graph and the resulting acceptance graph with an *inconclusive* verdict

As a consequence we calculate two types of observation sequences. One which could be calculated within the given bound, and one which could not. The latter type could be interpreted as follows: all observations made so far are correct, but not all observations could be calculated. For the test execution and evaluation this means that after processing all calculated observations, we have no further observations to which we can compare the remaining outputs of the SUT. We can neither say that further observations are correct nor can we say that they are not. We can only stop testing the SUT with this input sequence and give an *inconclusive* test verdict. This verdict says that all observations so far are correct but that we stopped further processing the current execution path. It would also be possible to decide for a *pass* or a *fail* verdict. But introducing a third verdict allows a finer distinction of differently caused test execution results. Hence, we distinguish two sets of possible observation sequences and the acceptance graph we build out of these sets comprises two accepting nodes. One for all observation sequences which could completely be generated and one for all observation sequences which were bounded. The acceptor itself is a deterministic finite automaton accepting both sets of observation sequences. A test case execution finishing in one of these nodes results in a pass or an inconclusive verdict. All observations not covered by the acceptance graph result in a fail verdict. On the right side of Fig. 4 you can see an acceptance graph for the execution graph on the left side.

A test case comprises the input sequence to stimulate the system under test and an acceptance graph to automatically evaluate the execution of this test case. The length of a test case and the number of test cases can be influenced by the selection policy of input sequences as explained above. The generated test suite is *sound*. That means that no correct systems under test will be rejected due to a test case. Instead, the test verdict fail will only be assigned if the observation of the system under test cannot be explained by the possible correct observations of the specification (see the conformance relation for state machines). This is true because we calculate all possible execution paths to generate the sets of possible correct observations. With unlimited computation power and time the presented algorithm is able to compute a *complete* test suite, which is capable to exactly differentiate between correct and incorrect implementations.

Algorithm 1 shows the control structure of the test case generation algorithm. The loop will be executed as often as inputs should be sent to the system under test in the

```
input  : state machine: sm
output  an acceptance graph
:
sm.configuration ← initial configuration
result ← initial simulation node
inconclusives ← ∅

while |trigger| < input length do
    trigger ← generate a new trigger
    store ← ∅

    forall node ∈ result do
        node.queue ⊕ trigger
        store ∪ {node}

    steps ← 0
    while result ≠ ∅ ∧ steps < limit do
        temp ← simulationStep(result)
        steps ← steps + 1, result ← ∅

        forall node ∈ temp do

            if steps = limit then
                inconclusives ∪ {node }
            else
                store ∪ {node}
                result ∪ {node}

    result ← store;
generateAcceptanceGraph(result,inconclusives)
```

Algorithm 1. Test case generation: control structure

test case. The inner while-loop controls the fix-point calculation of reachable status. While there are newly generated status the simulation step is successively repeated to calculate all reachable status. If there are no newly generated status the algorithm proceeds with the next input event. The results of the loop are a set of all completely calculated observation sequences and a set of all incompletely calculated observation sequences. Out of these sets an acceptance graph will be calculated.

Algorithm 2 shows the calculation of the successive status for the calculated status in the previous step. First, the state machine is initialized with the configuration from the status and the next trigger event is selected from the corresponding event queue. Then, all possible FTSs and all possible transition execution orders are executed to estimate the resulting status and the generated events. This includes: saving reached configuration, adding internal events to the input queue, and saving generated events which should be sent to the environment. The latter events are the possible correct observations which we use to build the acceptance graphs. Both the successive status and the generated events will be stored in a new simulation node. The set of all new simulation nodes will be returned as the result of the *simulationStep*.

The presented algorithm has exponential complexity. This complexity arises from the branch factor introduced by the different sets of firing transitions, the different possible execution orders of transitions, and the necessity to consider possible interleavings in

```
input  : set of simulation nodes: input
output  set of new generated simulation nodes: result
:
result ← ∅
forall node ∈ input do
    if node.queue ≠ <> then
        sm.configuration ← node.configuration
        event ← node.dequeue
        forall T_‖ : sm.getFTS(event) do
            permutations ← permute(T_‖)
            forall firing_transitions: permutations do
                effects ← []
                forall t: firing_transitions do
                    effects ← fire(t)
                temp ← node
                temp.configuration ← sm.configuration
                forall effect: effects do
                    forall ev: effect do
                        if ev ∉ E_SM then
                            temp.observation ⊕ ev
                        else
                            temp.queue ⊕ ev;
                result ∪ {temp }
                sm.configuration ← node.configuration
return result
```

Algorithm 2. Test Case Generation: Simulation Step

the event queue (asynchronous event processing and non-observable event store). The effort to calculate a test case grows with the length of an input sequence $\tilde{x}$ and indirectly by the number of internally generated events (expressed as a functional relation: $f(\tilde{x})$). The branch factor is bounded by the finite number of transitions and the finite number of events (c). Thus we can approximate the effort A to generate a test case for a given input sequence of length x as follows:

$$A(x) \sim e^{c \cdot (x + f(\tilde{x}))} \tag{10}$$

This exponential effort is visualized in the left diagram in Figure 5 by the gray doubly dotted graph. Due to the character of state machines this exponential effort cannot be avoided when pre-calculating test oracles. To weaken this problem we are also working on strategies to split input sequences and to combine test cases, respectively.

3.4 Combining Test Sequences

When testing non-terminating reactive systems it is also interesting to execute longer input sequences. To reduce non-determinism in the specification is not possible without any further knowledge about the system under test. Thus we concentrate on the

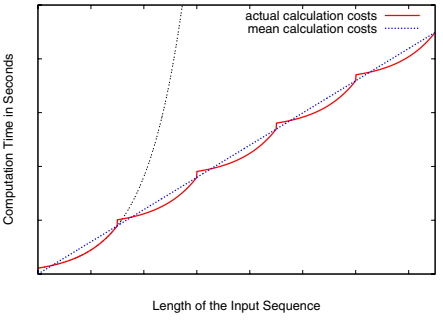


Fig. 5. Linearization of the exponential Complexity

asynchronous event processing. The lion's share of the calculation effort results from respecting all interleavings of the input sequence with internal generated events. We can argue that it is not necessary to consider all of these interleavings. For example, in practice it is the case that the system under test immediately starts to process the first received input. It usually does not wait until "ten" events are received from the environment. With the distance of two events in the input queue the probability decreases that an internally generated event (as a consequence of processing the first event) interleaves with the second one.

Based on this idea we developed various strategies to reduce the calculation effort. To demonstrate the core idea we implemented a strategy where we introduce so called *observation points*. Observation points are points in time where we give the system under test enough time to calculate its reaction. Related to our semantic model of state machines the system under test reaches a status in which the event queue is empty. Hence, no more reaction can be produced for the given inputs. This is true for all status at the hull of the execution graph from the previous section. Continuing after such an observation point now means: to enqueue the next input to all (non-inconclusive) status on the hull of the previously calculated execution graph (note that for these status the event queue is empty). We also reset the collected possible observations and calculate the corresponding acceptance graph. This is possible because we assumed that the system under test has completely calculated its reactions. Now we proceed to calculate the possible correct observation sequences for the complete next input sequence. An improvement of this strategy is to collect possible correct observations for more than one observation point and then generate one acceptance graph for all input sequences.

The reduction in the computation effort results from the fact that we do not consider possible interleavings resulting from events in the previous input sequence with events in the next input sequence. The left diagram in Figure 5 visualizes this procedure. We repeatedly calculate only the first part of the exponential curve. The overall calculation effort follows from adding the efforts needed to calculate the observations for the individual input sequences (the (red) solid graph). The average effort has a linear gradient depicted by the (blue) dotted graph. Compared to the effort for processing one input sequence with the length of the sum of all sub-sequences this is an enormous reduction in the calculation effort. The effort for combined test sequences still grows exponentially

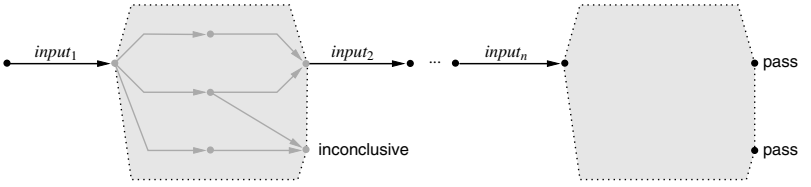


Fig. 6. General structure of a combined test case

with the length n of the particular input sequences but linear with the number x/n of combined sequences and consequently with the length x of the overall input sequence:

$$A_{comb}(n,x) \sim \frac{e^{c\cdot(n+f(\tilde{n}))}}{n}\cdot x \tag{11}$$

As a consequence of generating multiple acceptance graphs we would over-approximate the possible correct behavior. That means that we consider more observation sequences to be correct. This follows from the fact that observation sequences from different acceptance graphs can be combined in any possible order. This would not be possible for a complete input sequence. Consequently, the generation of an acceptance graph should be delayed as much as possible (e.g., in relation to the memory consumption). The generated test cases are still sound if the introduction of observation points is valid for the SUT.

Depending on the used testing strategy we can parameterize how test cases should be generated and combined. On the one hand by the effort we need to process the total count of inputs, and on the other hand by the reduction capability when splitting the input sequence into smaller parts. Figure 6 shows the structure of a test case with multiple input sequences and corresponding acceptance graphs. When reaching a pass node we continue to trigger the SUT with the next input sequence and check the newly generated output of the SUT at the next observation point. Experiments with this "static" strategy showed that if we can introduce such observation points for the system under test this strategy works quite well. But we also work on more elaborate "dynamic" strategies (e.g., to take advantage of specific properties of used events of the event store, probabilistic strategies to specify possible event interleavings, or memory and time consumption).

3.5 Evaluating the Test Process

If a test suite is generated with the algorithm above and if a SUT is tested with this test suite we would like to know how extensively we tested the system under test. The number of test cases and the length of the input sequences in the test cases only conditionally allow to draw conclusions related to that question. Still today the question is hard to answer. The mostly used approach is to measure the coverage of different elements of the system under test or the specification. For program code this is common practice. The used criteria are usually based on control flow or data flow information in the code or on functional description in the specification. With our test approach we

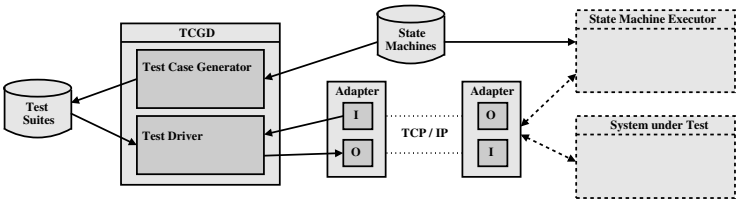


Fig. 7. Architecture of the TEAGER tool suite

address embedded reactive systems composed of hardware and software components. You can apply well known techniques to measure coverage in the software components, but our impression is that this is not sufficient for such systems. To measure coverage in the hardware components is usually not possible. The only way to regard the whole system is to use the specification. In further work we develop meaningful criteria for state machines. Our current work introduces different criteria based on structural elements of state machines, like states and transitions, and on semantic elements, like configurations and sets of firing transitions. Especially semantic criteria are able to evaluate the behavior in a more meaningful manner. An interesting question is whether it is possible to use such criteria to control the test case generation process (i.e., to measure coverage while generating test cases and to select the next inputs according to this coverage).

4 Experimental Results

To evaluate our complete test approach we implemented the TEAGER tool suite [29]. TEAGER consists of an environment to automatically generate and execute test cases, and additionally of an environment to execute state machine specifications. We use the latter to analyze the execution behavior and the testability of a state machine, and to measure coverage on a state machine specification to evaluate generated test suites. Figure 7 shows this general architecture. We us the TEST CASE GENERATOR to automatically generate test cases out of a state machine specification. A state machine specification is executed to compute the possible correct observation sequences for selected inputs. Based on them an acceptance graph is generated as the test oracle. Input sequences and acceptance graphs are stored for each test case in separate files for later execution. The TEST DRIVER in turn loads saved test cases and executes them. The execution includes both: stimulating the system under test and comparing the observation to the computed possible correct behavior in the acceptance graphs. The communication with the system under test takes place over a socket connection using pre-implemented adaptors. This concept offers a flexible way to connect the system under test. It also offers the possibility to use our STATE MACHINE EXECUTOR as a system under test stub. Thus we can analyze the execution behavior of state machine specification or measure the coverage of a used specification. The complete test case generation process is parameterized to have maximal control over the structure of test cases and the effort needed to calculate them. For more information about the TEAGER tool suite, its individual components, and the used parameters, we refer the interested reader to our web site (`swt.cs.tu-berlin.de/~seifert/teager.html`).

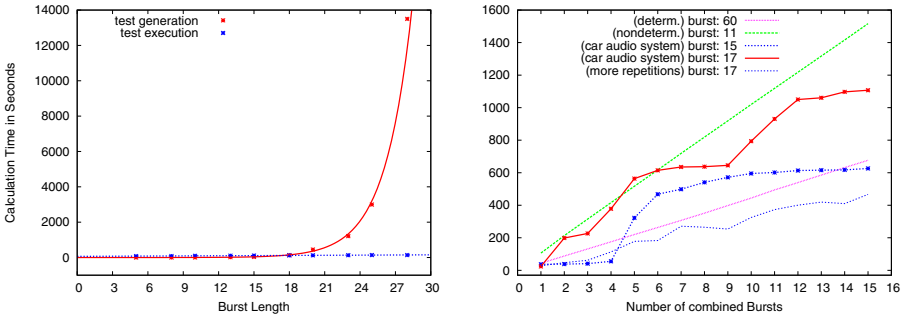


Fig. 8. Results for the first and second experiment

We applied two case studies on Pentium IV 2.6 GHz to evaluate our test case generation and execution. First, the Car Audio System from Section 2 and second, a system to control the sun blinds of an office building [25]. Generally speaking the results from both case studies allow the same interpretation. In the following, we briefly review some results from the Car Audio System case study to give an impression of the execution behavior of our test approach. We present two different experiments. In the first experiment we demonstrated the exponential calculation effort needed to calculate the possible correct observations. In the second experiment we demonstrated the effect of combining multiple input sequences. We used the state machine model from Figure 1 as specification. In all experiments we generated a test suite comprising 25 test cases. In the first experiment we varied the length of the input sequence and in the second experiment we fixed the length of the input sequences but varied the number of combined sequences.

Figure 8 illustrates the results of the two experiments. The (red) solid graph in the left picture clearly shows the exponential calculation effort of the test generation process. But it also shows that a relatively long input sequence can be processed even for a complex system. The (blue) dashed graph shows that executing a test case takes considerably less time, and that the time need only slightly increases the longer the input sequences are. From this it follows that the strategy to spend more time in test case generation to generate longer test cases and to save the test cases to be able to execute them multiple times is worthwhile. The (blue) dashed graph (burst size of 15) and the (red) solid graph (burst size of 17) in the right picture visualize the results of the second experiment. The unequal gradients of the subsections in the graphs result from the different calculation effort for the particular inputs (which were selected by a random strategy). To verify this we executed the experiments multiple times and calculated the mean values (the lower (blue) dashed graph). This graph shows that the generation times converge towards a linear graph. Additionally, we experimented with a deterministic and a non-deterministic specification (the green and magenta graph). The graphs clearly show the expected linear gradient. Finally we mention that the execution effort increases due to the higher number of observation points (which requires to wait for all system reactions). In practice, we need to choose an optimum with respect to the calculation effort and the execution time.

5 Summary and Outlook

Testing benefits from the fact that the actual system is brought to execution. Thus, the interaction of the real hardware and the real software can be evaluated. It is applicable at different levels of abstraction and at different stages of the development. It aims in falsification, that means to show inconsistencies between the specification and the developed system.

Our test approach allows to use UML state machines in quality assurance to precisely specify the reactive behavior of a system, and thus, to serve as the basis for the automated test case generation, execution and evaluation. To generate tests we select relevant input sequences and calculate the possible correct observation sequences for them. Based on these observations we calculate the test oracles which we use to automatically evaluate the test executions. Manually performed, this is a difficult and time consuming task. The approximation techniques we applied make the generation process efficient. It is possible to control the complete process via parameters depending on the time and computation power you want to invest. The modularization of the tasks gives our approach a clear structure and makes it interesting for further research. All discussed strategies are implemented as modules of the TEAGER tool suite. Thus, different strategies for selecting inputs, for combining test cases to reduce the calculation effort, or to select relevant data during test case generations can be studied independently from each other. Moreover, in practice this allows adaptation to different needs. We use a precisely defined semantics for UML state machines which includes complex structured data. We do not restrict state machines to ease test case generation. Instead, we follow the semantics description of the UML standard [1] as much as possible. Only misleading or conflicting statements are clarified. We address all semantic details which arise from the different sources of non-determinism. In particular we address the problem of asynchronous communication which is introduced by the *run-to-completion* semantics of state machines. Many real life systems can show such behavior.

Our ongoing research deals with a comprehensive integration of our approach into an UML-based development. In particular we address questions: how to combine our approach with a component-based development approach and how to combine our technics with other successfully applied testing technics. Furthermore we integrate more and more syntactical elements into our formal semantics and analyze their influence on the automated test generation process. Perspectively we address two challenges, namely specifying and testing timed behavior and considering complex data to generate "interesting" test cases. We also develop techniques to control and evaluate our automated processes. Measuring coverage, especially on the specification, is one step into this direction. We are analyzing criteria based on state machines and their semantic model.

References

1. UML2: Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, Version 2.1.1, formal/07-02-03 (2007), `http://www.uml.org/uml`
2. Balser, M., Bäumler, S., Knapp, A., Reif, W., Thums, A.: Interactive Verification of UML State Machines. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308. Springer, Heidelberg (2004)

3. Lilius, J., Paltor, I.P.: Formalising UML State Machines for Model Checking. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723. Springer, Heidelberg (1999)
4. Latella, D., Majzik, I., Massink, M.: Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker. Formal Aspects of Computing (1999)
5. De Nicola, R., Hennessy, M.C.B.: Testing Equivalences for Processes. Theoretical Computer Science (1984)
6. Brinksma, E.: A Theory for the Derivation of Tests. In: Protocol Specification, Testing and Verification. North-Holland, Amsterdam (1988)
7. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software–Concepts and Tools (1996)
8. Lee, D., Yannakakis, M.: Principles and Methods of Testing Finite State Machines - A Survey. In: Proceedings of the IEEE (1996)
9. Petrenko, A.: Fault Model-driven Test Derivation from Finite State Models: Annotated Bibliography. LNCS. Springer, Heidelberg (2001)
10. Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. LNCS. Springer, Heidelberg (2001)
11. Fujiwara, S., van Bochmann, G., Khendek, F., Amalou, M., Ghedamsi, A.: Test Selection Based on Finite State Models. IEEE Transactions on Software Engineering (1991)
12. Yang, B., Ural, H.: Protocol Conformance Test Generation using multiple UIO Sequences with Overlapping. SIGCOMM Computer Communication Review (1990)
13. Chow, T.S.: Testing Software Design Modeled by Finite-State Machines. IEEE Transactions on Software Engineering (1978)
14. Luo, G., van Bochmann, G., Petrenko, A.: Test Selection Based on Communicating Nondeterministic Finite State Machines Using a Generalized Wp-Method. IEEE Transactions on Software Engineering (1994)
15. Gnesi, S., Latella, D., Massink, M.: Formal Test-case Generation for UML Statecharts. In: Engineering Complex Computer Systems (ICECCS). IEEE Computer Society Press, Los Alamitos (2004)
16. Offutt, A.J., Liu, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specifications. Software Test, Verification. Reliability (2003)
17. Latella, D., Massink, M.: A Formal Testing Framework for UML Statechart Diagrams Behaviours: From Theory to Automatic Verification. In: International Symposium on High-Assurance Systems Engineering. IEEE Computer Society Press, Los Alamitos (2001)
18. Latella, D., Massink, M.: On Testing and Conformance Relations for UML Statechart Diagrams Behaviours. SIGSOFT Software Engineering Notes (2002)
19. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: The Unified Modeling Language (UML). Springer, Heidelberg (1999)
20. Pretschner, A., Lötzbeyer, H., Philipps, J.: Model based Testing in incremental System Development. Journal of Systems and Software (2004)
21. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. In: International Symposium on Software Testing and Analysis (ISSTA 2004), pp. 129–132 (2004)
22. Campbell, C., Grieskamp, W., Nachmanson, L., Schulte, W., Tillmann, N., Veanes, M.: Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. Technical Report MSR-TR-2005-59, Microsoft Research (2005)
23. Harel, D.: Statecharts: A Visual Formulation for Complex Systems. Science of Computer Programming (1987)
24. Seifert, D.: An Executable Formal Semantics for a UML State Machine Kernel Considering Complex Structured Data. Technical Report inria-00274391, DEDALE (LORIA) (2008)
25. Seifert, D.: Automatisiertes Testen asynchroner nichtdeterministischer Systeme mit Daten. Shaker Verlag, Also: PhD dissertation, Technische Universität Berlin (2007)

26. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. ACM Transactions on Software Engineering and Methodology (1996)
27. De Nicola, R.: Extensional Equivalences for Transition Systems. Acta Informatica (1987)
28. Seifert, D., Souquières, J.: Using UML Protocol State Machines in Conformance Testing of Components. Technical Report inria-00274383, DEDALE (LORIA) (2008)
29. Santen, T., Seifert, D.: Teager - Test Automation for UML State Machines. In: Software Engineering 2006. LNI, GI (2006)

An Approach to Testing with Embedded Context Using Model Checker

Lihua Duan and Jessica Chen

School of Computer Science, University of Windsor
Windsor, Ont. Canada N9B 3P4
{duan1,xjchen}@uwindsor.ca

Abstract. Testing each component in isolation is not always feasible. We consider FSM-based deterministic testing on an Implementation Under Test (IUT) together with some other correctly implemented components as its context. The behavior of the context needs to be taken into account for generating test sequences. We employ model checking tools to retrieve necessary information from the context specification so that a test suite for the IUT integrated with its context can be generated. The use of model checking tools frees us from the necessity of constructing the global model of the IUT and its context, and thus helps avoid the state explosion problem. In the current work, we consider the situations when the context is an embedded system, i.e. it communicates and only communicates with the IUT. In this setting, we present a method to derive a *complete* test suite that can be used to check for trace pre-order between the FSM representing the integrated implementation of the IUT and its context and the synchronous product of the specification FSM of the IUT and that of its context.

Keywords: finite state machines, conformance testing, context-based testing, test sequences, distinguishing sequences.

1 Introduction

Given a final software product, we are interested in knowing whether it *conforms* to what we expect. *Conformance testing* has been extensively studied and has turned out to be an effective tool for us to gain enough confidence in the correctness of our product implementation with respect to the expectations. It has two main characteristics: (i) Different from a formal verification approach, here the *implementation* (called *implementation under test* (IUT)) is treated as a *black box* from which we can only infer its behavior by providing input to it and observing its output. (ii) Instead of simple input/output pairs, the expected behavior in different states is formally specified. This is because in most of the cases, the given IUT is *stateful* in the sense that it reacts differently (e.g. by giving different outputs) to the same input provided at different time of the execution.

There are various tools to describe the expected behavior in different states. Suitable for different levels of abstractions, they range from formal specification languages such as process algebras, to structural/operational modelling

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 66–85, 2008.

languages such as (input/output) *labelled transition systems (LTSs)* and *Finite State Machines (FSMs)*. Of course, when a specification is given in some high level formal specification languages, we may still refer to its formal model.

Given a formal model $\mathcal{S}$ in terms of e.g. LTS or FSM, describing the expected behavior of the IUT, we can imagine that the IUT behaves according to a certain abstract machine $\mathcal{M}$ in the same format. In this setting, conformance testing amounts to establishing the correspondence between $\mathcal{S}$ and $\mathcal{M}$. There are several relations proposed in the literature in this regard: the *trace equivalence* relation, the *ioco conformance* relation [1], the *quasi-equivalence relation* [2], etc. In this work, we consider *trace pre-order* $\preceq$. $\mathcal{S} \preceq \mathcal{M}$ holds if any (input/output) trace allowed by $\mathcal{S}$ are implemented, yet a trace not specified in $\mathcal{S}$ may or may not be implemented. In a special case when the specifications of the IUT and its context are both completely specified, our results may hold for *trace equivalence*.

There is usually an infinite number of traces in a given model of the desired behavior and the one representing the IUT, some of them with infinite lengths. The ultimate goal for test generation is to find a sufficient and efficient set of finite input sequences, i.e. a test suite, from a given model so that when these input sequences are given to the IUT, we can, by comparing the actual output sequences with the expected ones, draw a conclusion whether the trace pre-order holds between the model of the expected behavior and the one representing the IUT.

This goal can be better achieved with the *slow environment* assumption well-adopted in the literature, i.e., whenever an input reaches the system, the system will always prompt the output for it and reach a *stable state* (i.e. there is no more executable statement from that state) before the next input can reach the system. In other words, each input is explicitly associated with one or more outputs. In this setting, under the assumption that the given model and the one representing the IUT share the same sets of inputs and outputs, we may be able to *identify* a state in the model that represents the IUT by observing a sequence of outputs in response to a special sequence of inputs. In protocol testing, people used FSMs to represent the state changes with the paired input and output, and have explored the characteristics of such special input sequences as expressed in the notions of *characterization set* [3], *Unique Input/Output sequence* (UIO) [4], *distinguishing sequence* [5]. With a *characterization set*, a set of *UIO sequences*, or a *distinguishing sequence*, we can identify the states in the implementation FSM with those in the specification FSM, based on which we can further *verify* the correspondence of the transitions in the specification FSM with those in the implementation FSM. This helps us establish an equivalence or pre-order relation between the specification FSM and the implementation FSM.

One of the major drawbacks of FSM-based testing is that the specification FSM, possibly derived from a specification given in a higher level of abstraction, suffers from the state explosion problem: it may have too large a state space even if we only consider the *control data*. This is troublesome especially when we consider testing more general software systems than communications protocols. A promising solution to this problem is found in the compositional testing approach: we can apply FSM-based testing techniques for unit testing,

while leaving the correctness of the integrated system to formal verification. Of course, this is a sound approach only if the considered equivalence or pre-order relation is *compositional*, e.g., if the specifications representing components $\mathcal{I}_1$ and $\mathcal{I}_2$ are X-equivalent to their respective behavioral specifications $\mathcal{S}_1$ and $\mathcal{S}_2$, then without performing *integration testing* or *system testing*, we know that the integration of $\mathcal{I}_1$ and $\mathcal{I}_2$ is a correct implementation of the parallel composition of $\mathcal{S}_1$ and $\mathcal{S}_2$ with respect to this X-equivalence. As we know, trace equivalence and trace pre-order are compositional when $\mathcal{I}_1$ and $\mathcal{I}_2$ are sequentially executed components represented by *deterministic* FSMs. The condition for *ioco* conformance relation to be compositional is given in [6].

Along the compositional testing approach to applying FSM-based testing techniques to unit testing, we may still encounter difficulties. One of them comes from the fact that testing each component in isolation is not always feasible. There are situations when we have to test a component together with some others.

As pointed out in [2], this can be the case when the IUT is an embedded part of a complex *system under test*. As another example, suppose we want to test a web-based composite service implementation $\mathcal{I}_1$. $\mathcal{I}_1$ makes use of another component service $\mathcal{I}_2$ which is known to be correct. When testing $\mathcal{I}_1$ in isolation, we have the difficulty in providing input and observing output all encapsulated according to certain protocol such as SOAP. Testing $\mathcal{I}_1$ in isolation also invokes the necessity of testing the interoperability between $\mathcal{I}_1$ and $\mathcal{I}_2$. A better solution is to test $\mathcal{I}_1$ and $\mathcal{I}_2$ together where $\mathcal{I}_2$ is considered as the *context* of $\mathcal{I}_1$.

Following the framework presented in [2] on *testing in context*, we consider the problem of FSM-based deterministic testing on $(\mathcal{I}, \mathcal{I}_c)$ which is an IUT implementation $\mathcal{I}$ together with a correct context implementation $\mathcal{I}_c$. In our current work, $\mathcal{I}_c$ is an embedded system, i.e. it does not communicate with any component other than $\mathcal{I}$. The communication port between $\mathcal{I}$ and $\mathcal{I}_c$ is not *controllable* but *observable*. This means that the tester can neither provide input to the IUT using this port nor stop an input from the context to the IUT. It can, however, observe all the input from and all the output to the context. The specification of $\mathcal{I}$ is given in terms of a *2-port FSM*, one port for communicating with its context and one for its input/output with the environment/tester. The specification of $\mathcal{I}_c$ can be given in terms of either a specification language or a structural modelling language. We present a method to generate a suitable test suite that can be used to test $(\mathcal{I}, \mathcal{I}_c)$. More precisely, we modify the well-known W-method [3] to construct test sequences in order to establish trace pre-order between the FSM representing $(\mathcal{I}, \mathcal{I}_c)$ and the product of the specification FSM of $\mathcal{I}$ and that of $\mathcal{I}_c$. The ultimate goal of our work is to avoid generating the operational model of the give specification of $\mathcal{I}_c$ (if a higher level specification is provided) and constructing the global model of $\mathcal{I}$ and $\mathcal{I}_c$. In order to do so, we employ model checking tools to retrieve necessary information from the context specification so that test sequences for $(\mathcal{I}, \mathcal{I}_c)$ can be generated. In this way, we avoid the notorious state explosion problem. Note that it is straightforward to extend our work to a more general case where the embedded context consists of a set of components, each having its own port to communicate with $\mathcal{I}$.

The rest of the paper is organized as follows. In Section 2 we give a brief notational background introduction to FSM and FSM-based testing that will be used later on. Our problem is explained in detail in Section 3, followed by a guideline of a possible solution. How to use model checking tools to generate a complete test suite is presented in Section 5. To better illustrate our method, we give a running example in Section 6. At the end, we position our work among other pieces of related work, and conclude ours with some final remarks.

2 Notational Background

In this section, we introduce the preliminary notations and terminologies on n-port finite state machines and test sequence construction. They will be used later in this paper.

2.1 *n*-Port Finite State Machines

As we mentioned in the Introduction, we assume that the specification $\mathcal{S}$ of the IUT is given in terms of a 2-port FSM. A deterministic *n-port Finite State Machine* (also called *finite state machine* for short) is defined by a tuple $(S, I, O, \delta, \lambda, s_0)$.

- S is a finite set of states where $s_0 \in S$ is its initial state.
- $I = \bigcup_{i=1}^{n} I_i$, where I_i is the input alphabet of port i $(i = 1, \ldots, n)$. Being abstract, these input symbols encapsulate the information of the communication channels. Thus, without loss of generality, we can assume that the input symbols at different ports are distinct, i.e. $I_i \cap I_j = \emptyset$ for $i \neq j$.
- $O = \Pi_{i=1}^{n} O_i$ where O_i is the output alphabet of port i $(i = 1, \ldots, n)$. Each $o \in O$ is a *vector of outputs* denoted by $o = \langle o_1, \ldots, o_n \rangle$ where $o_i \in O_i$ for $i = 1, \ldots, n$. We do not consider the order in which we observe output o_i and o_j at different ports. When there is no output at a port i, we use a special and distinct symbol $-$ to denote it.
- δ is the transition function that maps $S \times I$ to S, and λ is the output function that maps $S \times I$ to O.

The input and output symbols are abstract: The discussions on data types and complicate data structures in the input and the output are not considered.

Note that λ and δ are partial functions. We will use $\delta(s, i) = \mathit{null}$ to denote that there is no image of δ for the given state s of S and the given input i of I. In this case, we also have $\lambda(s, i) = \mathit{null}$. Furthermore, we extend the input of λ and δ from an input alphabet to a sequence of input alphabets with their meanings obtained straightforwardly from the original ones.

A *transition* t is defined by a tuple $(s_1, s_2, i/o)$ in which s_1 is the *starting state*, i is the input, $s_2 = \delta(s_1, i)$ is the *ending state*, and $o = \lambda(s_1, i)$ is the output. The input/output i/o is called the *label* of t. We use T to denote the set of all transitions in $\mathcal{S}$.

Let t_i be a transition for $1 \leq i \leq k$. A *path* $\rho = t_1\ t_2\ \ldots\ t_k$ is a finite sequence of transitions such that for $k \geq 2$, the ending state of t_i is the starting state of

t_{i+1} for all $1 \leq i \leq k-1$. A state $s \in S$ is *reachable* if there exists a path starting from s_0 and ending at s. We consider FSMs where all states are reachable.

An FSM is *completely specified* if functions λ and δ are *total*; otherwise, it is *partially specified*. We consider that the given specification FSMs are partially specified. Results in our approach when a given specification is completely-specified are discussed at the end. Note that if an FSM $\mathcal{S}$ is not completely specified, it is possible to make $\mathcal{S}$ completely specified by adding transition $(s, s, i/\langle -, \ldots, - \rangle)$ for each $s \in S, i \in I$ such that $(s, i) \notin domain(\delta)$. This, however, slightly changes the meaning of the FSM and is not always acceptable.

Two states s_i and s_j are *equivalent* if, for every input sequence σ, $\lambda(s_i, \sigma) = \lambda(s_j, \sigma)$. If $\lambda(s_i, \sigma) \neq \lambda(s_j, \sigma)$ then σ *distinguishes* between s_i and s_j. An FSM M is *minimal* if every state can be reached from the initial state of $\mathcal{M}$ and no two states of $\mathcal{M}$ are equivalent. Since only deterministic FSMs are considered, we can easily obtain a minimal FSM from any given FSM. In the following, we assume that all given FSMs are minimal.

2.2 Distinguishing Sequence and Test Sequence Construction

Let $\rho = (s_1, s_2, i_1/o_1)(s_2, s_3, i_2/o_2) \ldots (s_k, s_{k+1}, i_k/o_k)$ $(k \geq 1)$ be a path in an FSM. We will use $is(\rho)$ to denote the input sequence $i_1 \circ i_2 \circ \ldots i_k$ of ρ. Note that for clarity, we use $\circ$ as a separator in a sequence of input, a sequence of output, or a sequence of input/output pairs. A *test sequence* is an input sequence and is typically obtained from a path of a given specification FSM. A test suite is a finite set of finite test sequences. Usually, we assume that the IUT can always be reset to its initial state from any state and thus a test suite refers to a set of input sequences derived from paths that start from the initial state s_0. In this way, we can carry out the test with the test sequences in a test suite one by one. For each input sequence σ in a test suite, we will use $path(\sigma)$ to denote the path that σ is derived from. We will also use $out(\sigma)$ to denote the expected output sequence which is actually $\lambda(s_0, \sigma)$.

Given an FSM $\mathcal{S}$, we are interested in the so-called *complete test suites w.r.t. trace pre-order*. That is, by applying its input sequences to the IUT and comparing the output sequences with the expected ones, we can distinguish any implementation FSM $\mathcal{M}$ of the IUT if $\mathcal{S} \not\preceq M$.

Since $\mathcal{S}$ and $\mathcal{M}$ are deterministic and minimal, this can be achieved by establishing correspondence between the states in $\mathcal{S}$ and those in $\mathcal{M}$. Then, for each transition $t = (s_1, s_2, i/o)$ in $\mathcal{S}$, we construct a test sequence to verify that there exists a transition $t' = (r_1, r_2, i/o)$ in $\mathcal{M}$ which starts from a state corresponding to s_1, ends at a state corresponding to s_2, and gives the same output o upon the same input i.

The states in the implementation FSM can be identified via *distinguishing sequence*, *Unique Input/Output sequence (UIO)*, or *characterization set*. State identification using UIOs is possible but it turns out to be hard and less practical [7]. A characterization set is easier to find than a distinguishing sequence, yet a test suite generated using a characterization set [3] is usually much longer than that generated using a distinguishing sequence in terms of total length of the test sequences [5, 8–10]. Here we consider using distinguishing sequence.

A *distinguishing sequence* is an input sequence D with the following characteristics: the output sequences produced by $\mathcal{S}$ in response to D in different states of $\mathcal{S}$ are all different, i.e., for all $s_i, s_j \in S$, if $s_i \neq s_j$ then $\lambda(s_i, D) \neq \lambda(s_j, D)$.

There are various methods proposed in the literature for generating test sequences in order to check if an IUT conforms to a given specification FSM. See [11] for a survey on this topic. A basic idea of constructing a complete test suite $\mathcal{T}$ w.r.t. trace pre-order with a given distinguishing sequence D can be described as follows.

- For each state s_k in specification FSM $\mathcal{S}$, find a path ρ_k starting from s_0 and ending at s_k, and add a test sequence $is(\rho_k) \circ D$ to $\mathcal{T}$. If the IUT passes test sequence $is(\rho_k) \circ D$, i.e. its output sequence in response to this input sequence is correct, then we say that the state of the IUT after applying $is(\rho_k)$ *corresponds to* s_k. As we assume that the number of states in the implementation FSM $\mathcal{M}$ is no more than that in $\mathcal{S}$, set $\{is(\rho_k) \circ D \mid s_k \in S\}$ actually helps us to establish a one-to-one correspondence between the states in $\mathcal{S}$ and those in $\mathcal{M}$.
- For each transition $(s_1, s_2, i/o)$ in $\mathcal{S}$, add test sequence $is(\rho_1) \circ i \circ D$ to $\mathcal{T}$. Since $is(\rho_1) \circ D \in \mathcal{T}$, we know that the state of $\mathcal{M}$ after applying $is(\rho_1)$ corresponds to s_1. Thus, $is(\rho_1) \circ i \circ D$ helps us to check that there exists a transition $(r_1, r_2, i/o)$ in $\mathcal{M}$ where r_1 and r_2 correspond to s_1 and s_2 respectively: We can verify whether the state of $\mathcal{M}$ after applying $is(\rho_1) \circ i$ corresponds to s_2 by applying D on it. This is because from the first step, we have used the same distinguishing sequence D to identify all the states in $\mathcal{M}$. We say that transition $t = (s_1, s_2, i/o)$ is *verified* in a test suite $\mathcal{T}$ if there exists an input sequence σ such that $\sigma \circ D \in \mathcal{T}$, $\sigma \circ i \circ D \in \mathcal{T}$ and $path(\sigma)$ is a path in $\mathcal{S}$ from s_0 to s_1.

This is actually a variation of Chow's W-method [3] in the case when (i) the number of states in the implementation FSM is no more than that in the specification FSM; and (ii) a distinguishing sequence rather than a characterization set is available. Thus, it is straightforward that a test suite such constructed is complete with repect to trace pre-order.

In the following, we present an extension of this method for testing in context.

3 Problem Description

As noticed in [2], testing an IUT in isolation is quite different from testing it within a context. First of all, if an IUT is tested within a context and passed a test, we cannot draw any conclusion about the correctness of the IUT because a fault in the IUT and a fault in its context may mask each other resulting in an overall correct execution. Such a problem is out of the scope of our current work. In the following, we consider that the context is correctly implemented, with its behavior specified in $\mathcal{C}$.

Recall that in our setting, the FSM for an IUT has two ports: one for communicating with its context, called the *context port*; and the other for communicating

with the rest part of its environment simulated by a tester, called *environment port*. For clarity, we will use

- I and O as the IUT's input and output at the environment port;
- X and Y as the IUT's input and output at the context port.

The behavior of the IUT is thus given as $\mathcal{S} = \langle S, s_0, I \cup X, O \cup Y, \lambda_s, \delta_s \rangle$.

We assume that the specification FSM $\mathcal{S}$ is free from *internal-port-cycles*. An *internal-port-cycle* in an FSM is a path $(s_1, s_2, i_1/o_1)\ (s_2, s_3, i_2/o_2)\ \ldots\ (s_k, s_{k+1}, i_k/o_k)$ $(k \geq 2)$ such that $s_1 = s_{k+1}$, and $i_j \notin I$ for all $1 \leq j \leq k$. An *internal-port-cycle* represents a possibly infinite internal communications between the IUT and its context, which is normally considered as a design error. How to guarantee that the design specifications are free from such logical errors can be carried out by formally verifying the correctness of the design specifications.

An input sequence generated from $\mathcal{S}$ cannot be served as an input sequence to test the IUT in its context $\mathcal{I}_c$, as we cannot control the IUT's context port. To take the context into consideration, a possible approach is to develop a testing technique to check whether $\mathcal{M}$ conforms to $\mathcal{S}$ *within context* $\mathcal{C}$ w.r.t. trace pre-order, instead of checking whether $\mathcal{M}$ conforms to $\mathcal{S}$ w.r.t. trace pre-order. That is, we compare the model representing the actually behavior of $(\mathcal{I}, \mathcal{I}_c)$ with the one specifying its expected behavior. Just like we assume that the actual behavior of the IUT can be described by an FSM for testing the IUT in isolation, we assume that the actual behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be described by an FSM.

The model representing the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be derived from the specification of the IUT and that of the context. Suppose that the context specification $\mathcal{C}$ is given as a 1-port FSM. Of course, if it is given in a specification language with higher level of abstraction, we consider its equivalent FSM model. Let

$$\mathcal{C} = \langle C, c_0, \bar{Y}, \bar{X}, \lambda_c, \delta_c \rangle$$

be the specification FSM of the context where $\bar{X} = \{\bar{x} \mid x \in X\}$ and $\bar{Y} = \{\bar{y} \mid y \in Y\}$ are the output and input symbols of $\mathcal{C}$ to communicate with $\mathcal{S}$: $\bar{x}$ and $\bar{y}$ are executed simultaneously with x and y respectively, representing the communications between the IUT and its context. Here we have ignored those actions internal to the context component.

Note that since we have the slow environment assumption, it makes no difference to use synchronous or asynchronous communication mode between the IUT and its context. For simplicity, we consider synchronous communication.

Given $\mathcal{S}$ and $\mathcal{C}$ as the above defined 2-port and 1-port FSMs, the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$ can be described as a synchronous product FSM $\mathcal{S} \times \mathcal{C}$ defined on $\mathcal{S}$ and $\mathcal{C}$ as $\langle S', (s_0, c_0), I, ((O \times Y) \cup X)^*, \lambda, \delta \rangle$. It has only one port with the tester/environment for input. A global state consists of a local state of $\mathcal{S}$ and a local state of $\mathcal{C}$. $S' \subseteq S \times C$ is a set of global states reachable from (s_0, c_0) in the sense that for any $(s, c) \in S'$, there exists an input sequence $\sigma \in I^*$ such that $\delta((s_0, c_0), \sigma) = (s, c)$.

$((O \times Y) \cup X)^*$ is a set of outputs from the tester's viewpoint. As we mentioned in the Introduction, we assume that even though the input/output between the

IUT and its context is not controllable, they are observable. Thus, corresponding to each input from the environment, the tester will observe a sequence of outputs which is composed of those outputs $\langle o, y\rangle$ of the transitions in $\mathcal{S}$ ($\langle o, y\rangle \in O \times Y$) and those input x from its context ($x \in X$).

A transition in $\mathcal{S}\times\mathcal{C}$ is derived from a path in $\mathcal{S}$ and a path in $\mathcal{C}$. More precisely, we have transition $((s_1, c_1), (s_2, c_2), i/o)$ in $\mathcal{S} \times \mathcal{C}$, and thus $\lambda((s_1, c_1), i) = o$ and $\delta((s_1, c_1), i) = (s_2, c_2)$, only if we have

$$\lambda_s(s_1, i_1 \dots i_k) = o_1 \dots o_k,\ \delta_s(s_1, i_1 \dots i_k) = s_2,$$

$$\lambda_c(c_1, i'_1 \dots i'_h) = o'_1 \dots o'_h,\ \delta_c(c_1, i'_1 \dots i'_h) = c_2;$$

for $h, k \geq 1$ such that

$$k = h,\ i = i_1,\ o = o_1 \circ i_2 \circ o_2 \dots \circ i_k \circ o_k,$$

$$i'_j = \overline{c(o_j)} \text{ for } 1 \leq j \leq k,\ i_{j+1} = \overline{o'_j} \text{ for } 1 \leq j \leq k-1,\ o'_k = -;$$

or

$$k = h + 1,\ i = i_1,\ o = o_1 \circ i_2 \circ o_2 \dots \circ i_k \circ o_k,$$

$$i'_j = \overline{c(o_j)} \text{ for } 1 \leq j \leq k-1,\ i_{j+1} = \overline{o'_j} \text{ for } 1 \leq j \leq k-1,$$

$$o_k = \langle *, -\rangle \text{ where * can be any output including -;}$$

Otherwise, $\lambda((s_1, c_1), i) = \mathit{null}$ and $\delta((s_1, c_1), i) = \mathit{null}$. Here $c(o)$ represents the output of o at the context port. Note that in the following, when there is no confusion, we will drop the subscripts of λ and δ.

Since there is no *internal-port-cycle* in $\mathcal{S}$, the above defined product FSM fully describes the expected behavior of the IUT with its context using the slow environment feature. Furthermore, as we assume that $\mathcal{S}$ and $\mathcal{C}$ are minimal and deterministic, the above defined synchronous product of them is also minimal and deterministic.

Once we have a product FSM specification for the expected behavior of $(\mathcal{I}, \mathcal{I}_c)$, it is straightforward to generate a suitable test suite from this product FSM in order to test whether trace pre-order holds between this specification and the implementation FSM of $(\mathcal{I}, \mathcal{I}_c)$.

This approach, however, requires that the FSM specification of $\mathcal{I}_c$ be available, and the global model of $(\mathcal{I}, \mathcal{I}_c)$ be calculated, which brings out the state explosion problem. In the present work, we consider using model checker as an auxiliary tool to retrieve necessary information from a context specification in order to generate test sequences. We do not require that the product of $\mathcal{S}$ and $\mathcal{C}$ be actually constructed. In particular, if the specification of the expected behavior of $\mathcal{I}_c$ is given in a specification language of a higher level of abstraction, we do not need to construct its operational model neither.

4 Test Generation with Context

To check whether a trace pre-order relation holds between $\mathcal{S} \times \mathcal{C}$ and the implementation FSM of $(\mathcal{I}, \mathcal{I}_c)$, according to what we introduced in Section 2, we

need to generate a complete test suite to *identify* all the states in $\mathcal{S} \times \mathcal{C}$ using a distinguishing sequence, and *verify* all the transitions in $\mathcal{S} \times \mathcal{C}$ using the same distinguishing sequence. Since the context implementation is known to be correct, we actually only need to generate test sequences to verify *some* of the transitions in $\mathcal{S} \times \mathcal{C}$. Consequently, we can look for a distinguishing sequence that is capable of distinguishing only a subset of states in $\mathcal{S} \times \mathcal{C}$. In this section, we characterize such a subset of transitions and a subset of states.

Definition 1 ($\mathcal{R}$ covers T). *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R} \subseteq T$. $\mathcal{R}$ covers T if for any transition $((s_1, c_1), (s_2, c_2), i/o) \in T$, there exists a transition $t = ((s_1, c_1'), (s_2, c_2'), i/o)$ in $\mathcal{R}$ where $(s_1, c_1), (s_2, c_2), (s_1, c_1')$, and (s_2, c_2') are states in $\mathcal{S} \times \mathcal{C}$, i is an input of $\mathcal{S} \times \mathcal{C}$ and o is an output of $\mathcal{S} \times \mathcal{C}$.*

The transitions in $\mathcal{S} \times \mathcal{C}$ can be partitioned into different groups according to the local states of $\mathcal{S}$ in their starting states, the local states of $\mathcal{S}$ in their ending state, and their input/output pairs. The above definition actually requires that the subset of transitions $\mathcal{R}$ contain at least one representative transition from each of the partitions. The intuition behind is this: Since $\mathcal{S}$ and $\mathcal{C}$ are deterministic, given two states s_1 and s_2 in $\mathcal{S}$, an input i and an output o in $\mathcal{S} \times \mathcal{C}$, there exists exactly one path ρ in $\mathcal{S}$ from s_1 to s_2 with input/output sequence $i_1/o_1 \circ i_2/o_2 \circ \ldots \circ i_k/o_k$ such that $i = i_1$ and $o = o_1 \circ i_2 \circ o_2 \circ \ldots \circ i_k \circ o_k$. According to the definition of synchronous product, for any states c_1, c_2 in $\mathcal{C}$, if transition $t = ((s_1, c_1), (s_2, c_2), i/o) \in T$, then t is constructed from this path. Consider all such transitions in one partition $G(s_1, s_2, i, o)$. To check that each transition in $G(s_1, s_2, i, o)$ is correctly implemented, we only need to make sure that path ρ is correctly implemented in the sense that there exists a path ρ' in $\mathcal{M}$ which starts from a state identified as s_1, ends at a state verified as s_2, and correctly gives output o in responds to input i. Since the context is correct, this implies that all transitions in partition $G(s_1, s_2, i, o)$ are correctly implemented. While any transition in $G(s_1, s_2, i, o)$ can be used to generate a test sequence for the above purpose, we require that the subset $\mathcal{R}$ of transitions contains one transition from each partition $G(s_1, s_2, i, o)$.

As we consider only transitions in such a subset of transitions $\mathcal{R}$ that covers the total set of transitions in $\mathcal{S} \times \mathcal{C}$, we only need a distinguishing sequence to identify all the states appeared as the starting or ending states in the transitions in $\mathcal{R}$, denoted by $states(\mathcal{R})$. In the following, we show that we can further weaken this requirement: it is sufficient to have a distinguishing sequence that can identify, among the states in $states(\mathcal{R})$, all those with different local states of $\mathcal{S}$.

Definition 2 (distinguishing sequence on $\mathcal{S}$ over $\mathcal{W}$). *Let $\mathcal{W}$ be a subset of reachable states in $\mathcal{S} \times \mathcal{C}$. An input sequence $D = i_1 \circ \tilde{x}_1 \circ i_2 \circ \tilde{x}_2 \ldots \circ i_k \circ \tilde{x}_k$ for $i_j \in I$, $\tilde{x}_j \in X^*$ $(1 \leq j \leq k)$ is a distinguishing sequence on $\mathcal{S}$ over $\mathcal{W}$ if*

- *For any state s, $s' \in S$, $s \neq s'$ implies $\lambda(s, D) \neq \lambda(s', D)$.*
- *For any $(s_1, c_1) \in \mathcal{W}$ and for any h $(1 \leq h \leq k)$, the input sequence of X^* obtained from $\lambda((s_h, c_h), i_h)$ by removing all output of Y is $\tilde{x}_h$. Here for $2 \leq h \leq k$, $(s_h, c_h) = \delta((s_1, c_1), i_1 \circ i_2 \ldots \circ i_{h-1})$.*

The above definition can be viewed as an extension of the normal definition of distinguishing sequence of an FSM: A distinguishing sequence of $\mathcal{S}$ over $\emptyset$ is actually the original definition of distinguishing sequence on $\mathcal{S}$ without considering any context.

Note that we do not require an input sequence to distinguish all the states in $\mathcal{S} \times \mathcal{C}$, but a subset of states of interest expressed in $\mathcal{W}$. This brings out two benefits: i) an increased possibility of the existence of a distinguishing sequence; ii) when there exist distinguishing sequences, a possibly shorter one which contributes to the reduction of the cost for carrying out the test.

Now we show that in order to generate from $\mathcal{S} \times \mathcal{C}$ a complete test suite w.r.t. trace pre-order, it is sufficient to consider a subset $\mathcal{R}$ of transitions as long as $\mathcal{R}$ covers its set T of transitions, and a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.

Note that while previous work on this topic for testing in isolation requires *reliable reset*, i.e. the IUT can be reset to its initial state at any time, here we assume that the IUT can be reset to its initial state at any time and its context will be reset at the same time.

Similar to previous work, we assume a bound on the number of states in the implementation FSM of the IUT. When we test an IUT with a context, since the input to the IUT from the context is not *controllable*, the description of the IUT can be considered as a 1-port FSM from the tester's viewpoint. As a consequence, some of the states in a given 2-port FSM are not *stable* (so-called *transient states* in [2]) in the sense that after an input from the tester/environment, the IUT will never stay in any of those states waiting for the next input from the tester/environment. For testing in context, we consider only stable states: When we say that the number of states in the implementation FSM of the IUT is no more than the number of states in the specification FSM of the IUT, we refer to those states that appear to be the starting states of some transitions with input at the environment port.

With the above assumptions, we present the following result:

Proposition 1. *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$ and $\mathcal{R} \subseteq T$. Let $\mathcal{T}$ be a test suite derived from $\mathcal{S} \times \mathcal{C}$. If*

- *$\mathcal{R}$ covers T,*
- *there exists an input sequence D such that D is a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$, and $\forall t = ((s_1, c_1), (s_2, c_2), i/o) \in \mathcal{R}$, there exists an input sequence σ such that $\sigma \circ D \in \mathcal{T}$, $\sigma \circ i \circ D \in \mathcal{T}$, and $\mathrm{path}(\sigma)$ is a path in $\mathcal{S} \times \mathcal{C}$ from (s_0, c_0) to (s_1, c_1),*

then $\mathcal{T}$ of $\mathcal{S} \times \mathcal{C}$ is complete w.r.t. trace pre-order.

The proof of this result is omitted due to the lack of the space and will appear in the full version of this work.

This proposition indicates that a desired test suite can be generated by finding a transition set $\mathcal{R}$ and a distinguishing sequence D such that $\mathcal{R}$ covers T and D is a distinguishing sequence over $states(\mathcal{R})$. In the next section, we will show how to find $\mathcal{R}$ and D with a model checker.

5 Test Generation Using Model Checking Tools

Model checking tools such as SPIN [12], SMV [13], UPPAAL [14] are originally designed to verify the correctness of design specifications. Recent years have seen trends in applying model checking tools to assist the test generation procedures (see e.g. [2, 15–19]). When we use a model checker to verify a system model against some required property, a counter-example will be returned if the system model is not correct w.r.t. the property being checked. Making use of this functionality of model checkers, we can characterize a desired test sequence as a property. We use a model checker to verify the negation of this property, called *trap property*, against a system specification. When this trap property is violated, a counter-example returned by the model checker actually serves as a desired test sequence. Following this line of research, we present here another example of using model checkers to generate test sequences in conformance testing with context.

To avoid constructing synchronous product of $\mathcal{S}$ and $\mathcal{C}$, the specifications of the IUT and its context are given to a model checker as a system specification. The specification FSM of the IUT can be straightforwardly translated into any formal specification language accepted model checking tools. For its context, we do not restrict it to be given in a particular specification language or a particular model, as long as it can be translated into a specification language accepted by the adopted model checker. In the following, we use *Spec* to denote the specification for the composition of the IUT and its context given in the specification language of the chosen model checker.

We explain below how to make use of the specification FSM of an IUT and a model checker (with *Spec*) to derive a test suite of the IUT and its context that is complete with respect to trace pre-order.

5.1 Finding Transitions in $\mathcal{R}$

As we explained in Section 4, we need to find a subset $\mathcal{R}$ of transitions in $\mathcal{S} \times \mathcal{C}$ such that $\mathcal{R}$ covers T where T is the set of transitions in $\mathcal{S} \times \mathcal{C}$. Since the synchronous product FSM for the IUT and its context is not available, we analyze $\mathcal{S}$ and derive $\mathcal{R}$ via a model checker. Fig. 1 shows an algorithm to use a model checker to determine a transition set $\mathcal{R}$ such that $\mathcal{R}$ covers T.

A path $\rho = (s_1, s_2, i_1/o_1) \circ (s_2, s_3, i_2/o_2) \circ \ldots \circ (s_k, s_{k+1}, i_k/o_k)$ in $\mathcal{S}$ is *composable* if $i_1 \in I$, $i_j \in X$ for $2 \leq j \leq k$, and $\delta(s_{k+1}, i) \neq null$ for some $i \in I$. According to the definition of synchronous product in Section 3, any transition $t = ((s, c), (s', c'), i/o) \in T$ is constructed from some composable path. On the other hand, not all composable paths in $\mathcal{S}$ can be used to define a transition in $\mathcal{S} \times \mathcal{C}$. Those that can be used to define a transition in $\mathcal{S} \times \mathcal{C}$ are called *executable paths*. Recall that transitions of T in partition $G(s, s', i, o)$ share the same local state s of the IUT in its starting state, the same local state s' of the IUT in its ending state, and the same input i and output o. Each executable path is actually uniquely used to define all transitions in one of the partitions.

Now, as we want to derive a set $\mathcal{R}$ of transitions that contains at least one (arbitrary) transition in each partition, we can use an executable path ρ in $\mathcal{S}$ to

1: **Input:** $\mathcal{S}$, *Spec*.
2: **Output:** a set V of pairs of transitions in $\mathcal{S} \times \mathcal{C}$ and input sequences in I^*, $\mathcal{R}$.
3: Let Φ contains all composable paths in $\mathcal{S}$;
4: Let $V = \emptyset$;
5: **for** each path ρ in Φ **do**
6: define a formula ϕ to express the non-existence of a path in *Spec* which contains a subpath which is equal to ρ when all its transitions from the context are ignored.
7: use model checker to verify formula ϕ in *Spec*;
8: **if** formula ϕ is violated **then**
9: add (t, σ) to V, where (i) $t \in \mathcal{S} \times \mathcal{C}$ is a transition derived by ρ and a path in $\mathcal{C}$ defined by the counter-example returned from the model checker; and (ii) σ is an input sequence in I^* derived from the counter-example that defines a path from (s_0, c_0) to the starting state of t;
10: **end if**
11: **end for**
12: Let $\mathcal{R} = \{t \mid (t, \sigma) \in V\}$;
13: return V and $\mathcal{R}$;

Fig. 1. Algorithm 1. To find a transition set $\mathcal{R}$

request the model checker to find an arbitrary transition of T that represents the partition uniquely determined by ρ. This can be done as follows: Use temporal logic formula to express such a property that there exists a subpath which is equal to ρ when all its transitions from the context are ignored. Request the model checker to verify the trap property, i.e. the negation of the above property. If ρ is used to define a transition t in $\mathcal{S} \times \mathcal{C}$, then the model checker will detect the violation of the trap property, returning a path in *Spec* from which we can derive a transition in the partition of ρ. Note that in addition to the transition in T, we also derive from the counter-example an input sequence in I^* which defines a path from (s_0, c_0) to the starting state of t. This input sequence will be used later on to construct a test suite.

As statically we do not know which composable path is executable, we simply ask the model checker to check all composable paths. If a composable path is not executable, the model checker will prove the trap property. In this case, we do not need to record any information.

Since $\mathcal{S}$ is finite and free from internal-port-cycles, the number of composable paths in $\mathcal{S}$ is finite and the computation of Φ is in polynomial time. Consequently, the time complexity of Algorithm 1 depends on that of the model checking algorithms used by the model checker. See e.g. [13] for the discussions on the complexity of model checking algorithms. In fact, optimization techniques of model checking have been well studied in recent years to enhance its applicability. Thus, the practicality of Algorithm 1 is endorsed.

According to Algorithm 1, we have the following result. Again, its proof is omitted due to the lack of the space and will appear in the full version of this work.

Proposition 2. *Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R}$ the set of transitions obtained from Algorithm 1. We have $\mathcal{R}$ covers T.*

5.2 Finding a Distinguishing Sequence

Algorithms for finding a distinguishing sequence of an FSM are well-discussed in the literature. See [11] for a good survey on this topic. However, finding a distinguishing sequence of an FSM in context is much more complicated. Due to the fact that a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$ must be calculated with both the specification of the IUT and that of its context, while synchronous product FSM of them is not available, we will apply model checker again. In [20], the authors presented an approach to generating a distinguishing sequence of an EFSM with UPPAAL model checker [14]. Here, we adopt the idea of this approach to generate a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.

1: **Input:** *Spec*, $\mathcal{R}$.
2: **Output:** a distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$.
3: **for** each state (s, c) in $states(\mathcal{R})$ **do**
4: create a variant of *Spec* with (s, c) as its initial state;
5: **end for**
6: create a *monitor* process to synchronize all variants in the sense that a variant can only accept an input if all others accept the same input simultaneously;
7: define a formula ϕ to express the property that there does not exist an input sequence such that the corresponding output sequences produced by any two variants with different local states of $\mathcal{S}$ as their initial states are all different;
8: request model checker to verify ϕ in *Spec*;
9: **if** model check detects a violation **then**
10: Let D be the input sequence derived from the counter-example returned by the model checker;
11: return D;
12: **else**
13: return "There does not exist any distinguishing sequence on $\mathcal{S}$ over $states(\mathcal{R})$";
14: **end if**

Fig. 2. Algorithm 2. To find a distinguishing sequence over $states(\mathcal{R})$

Fig. 2 shows an algorithm for this purpose. Initially, for each state $(s, c) \in states(\mathcal{R})$, we create a variant of $\mathcal{S}$ with s as its initial state and a variant of $\mathcal{C}$ with c as its initial state. Then by making use of a special *monitor* process, we request all the processes that represent these variants of $\mathcal{S}$ to synchronize all their actions on accepting input from both the environment port and the context port so that they will always accept the same input at the same time. For any two variants whose local states of $\mathcal{S}$ in their initial states are different, if the output sequences produced upon a same input sequence are all different, then the input sequence can be used as a desired distinguishing sequence D on $\mathcal{S}$ over $states(\mathcal{R})$.

As we know, not every FSM has a distinguishing sequence, In our setting, we cannot guarantee either their existence. However, as distinguishing sequences very often exist in real-life examples, the distinguishing sequences in our setting also exist in many application examples.

The problem of finding a distinguishing sequence is PSPACE-hard by itself [11]. Algorithm 2 reduces the problem to an application of model checking tools. This allows us to benefit from important features that they provide, such as the efficient partial order reduction and OBDD, and thus, reduce the actual cost for the computation.

Finally, with V and D, a test suite $\mathcal{T}$ is obtained: For each $(t, \sigma) \in V$, add both $\sigma \circ D$ and $\sigma \circ i \circ D$ to $\mathcal{T}$, where i is the input of t.

6 An Application

In this section, we use Inter-library Loan System (ILS) as a running example and we use SPIN [21] as a supporting model checker to show how to use the proposed technique to generate a complete test suite w.r.t. trace pre-order for testing in context.

SPIN targets the efficient verification of a system model against the required properties on-the-fly. Here, the system model is described in Promela [21] and the required system properties are often expressed in Linear Temporal Logic (LTL) formulas. As a matter of fact, a design specification expressed in many other specification languages such as FSM and EFSM can be easily translated into a Promela model.

A simplified ILS consists of two components: a borrowing library and a lending library. A user at the borrowing library can search a book in the lending library. When a book is found, the user can choose either to purchase the book or to issue a loan request. The lending library will always grant the purchase of the book; however, the allowance of the loan of the book depends both on the availability of the required book and on the length of the waiting list. There are three cases: i) if the book is available, the loan request will be granted; ii) if the book is unavailable but the waiting list is not full, the lending library will ask the user if he/she wants to make a reservation; and iii) if the waiting list is full, the lending library will tell the user that the book is unavailable.

Suppose that the borrowing library is the IUT and the lending library is its context. The specification $\mathcal{S}$ of the IUT has two ports: $portUser$ and $portContext$. Port $portUser$ represents the interface of the borrowing library with the environment/tester, and port $portContext$ represents the interface of the borrowing library with its context, the lending library. The semantics of service primitives used in ILS can be inferred by their symbolic representations. For example, $searchBook$ is an input primitives at $portUser$ to represent a user's action of searching a book; $loanAccptd$ is an input primitives at $portContext$ to represent that a user's request of a book loan is accepted.

Fig. 3 and Fig. 4 give the specification FSM $\mathcal{S}$ of the borrowing library and the Promela model of the lending library $\mathcal{C}$, respectively. Suppose that the number of available books is 3, and the length of the waiting list for a book reservation cannot exceed 3. Let T be the set of transitions in $\mathcal{S} \times \mathcal{C}$, and $\mathcal{R} \subseteq T$. To find $\mathcal{R}$ such that $\mathcal{R}$ *covers* T and to find a distinguishing sequence over $states(\mathcal{R})$, we need to translate FSM $\mathcal{S}$ and the behavior of a user of the ILS

into Promela processes. Thus, there are three processes in the Promela model of ILS: $User$, $Borrower$ and $Lender$, which represent the specifications of the environment/tester, the borrowing library, and the lending library, respectively. To establish the communication among these processes, there are four channels.

- $fromUser$: a channel through which $Borrower$ receives inputs from $User$;
- $ToUser$: a channel through which $Borrower$ sends outputs to $User$;
- $fromLender$: a channel through which $Borrower$ receives inputs from $Lender$;
- $ToLender$: a channel through which $Borrower$ sends outputs to $Lender$;

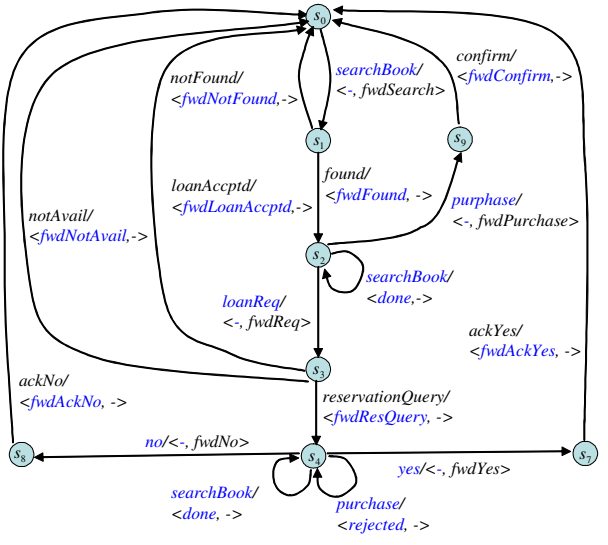


Fig. 3. Specification FSM of the borrowing library

Now we show how to find $\mathcal{R}$. Let $\rho = loanReq/\langle -, fwdReq\rangle \circ notAvail/\langle fwdNotAvail, -\rangle$. Clearly, ρ is a composable path in $\mathcal{S}$. In order to use SPIN to check whether ρ is executable, we need an LTL formula to express the negation of the existence of a transition in $\mathcal{S} \times \mathcal{C}$ derived from ρ.

Since the sending actions are always executable, we focus on finding a path to enable the receiving actions in ρ. Let the temporal logic variables be defined as follows:

$$r = Borrower@s_2$$

$$p = fromUser?[loanReq]$$

$$q = fromLender?[notAvail]$$

Here, r represents that process $Borrower$ is in state s_2; p represents that message $loanReq$ is received from channel $fromUser$; and q represents that message

```
proctype Lender() {
bool book; /*initialization*/
int inStock =3; /*No. of available books*/
int waitingLst = 0;
int Max = 3; /*the maximum length of waiting list*/

if
:: book = true;
:: book = false;
fi;

ac0: /*label ac0 is associated with abstract state ac0*/
if
:: book == true → toLender ? fwdSearch
                  → fromLender ! found;
:: book == false → toLender ? fwdSearch
                  → fromLender ! notFound;
                  goto ac0;
fi;

ac1: /*label ac1 is associated with abstract state ac1*/
if
:: toLender ? fwdReq;
if
:: inStock > 0 → fromLender ! loanAccptd;
                  inStock–;
                  goto ac0;
:: inStock <= 0 and waitingLst >= Max
                  → fromLender ! notAvail;
                  goto ac0;
:: inStock <= 0 and waitingLst < Max
                  → fromLender ! reservationQuery;
fi;
:: toLender ? fwdPurchase → fromLender ! confirm;
                  goto ac0;
fi;

ac2: /*label ac2 is associated with abstract state ac2*/
if
:: toLender ? fwdYes → fromLender ! ackYes;
                  waitingLst++;
                  goto ac0;
:: toLender ? fwdNo → fromLender ! ackNo;
                  goto ac0;
fi;
}
```

Fig. 4. Promela model of the lending library

notAvail is received from channel $fromLender$. Then the desired trap LTL formula can be expressed as

$$\phi = !(<> (rUp)Uq).$$

When verifying the ILS Promela model against ϕ, we obtain the following result from the returned counter-example:

$\sigma = searchBook \circ loanReq \circ searchBook \circ loanReq \circ searchBook \circ loanReq \circ searchBook \circ loanReq \circ yes \circ searchBook \circ loanReq \circ yes \circ searchBook \circ loanReq \circ yes \circ searchBook$

$t = ((s_2, c_{1,2}), (s_0, c_{0,4}), loanReq/\langle -, fwdReq\rangle \circ notAvail/\langle fwdNotAvail, -\rangle),$

where $c_{0,4}$ and $c_{1,2}$ are concrete states split from abstract state ac_0 and ac_1 in the situation when $inStock = 0$ and $waitingLst = 3$, respectively.

This result actually describes a possible scenario of having a transition in $\mathcal{S} \times \mathcal{C}$ derived from ρ when all the books in the lending library are checked out and the waiting list is full.

As shown in [11], the role of distinguishing sequences can actually be replaced by their prefixes, one for each state. This very often helps us achieve shorter test sequences. The definition of a distinguishing sequence over $\mathcal{W}$ can be extended to prefix distinguishing sequences D_i (for state s_i) straightforwardly. Following Algorithm 2, prefix distinguishing sequence D_i over $states(\mathcal{R})$ can be found with SPIN. For example, we have $D_0 = searchBook$ and $D_2 = D_4 = searchBook \circ purchase$. Thus, test sequences for t are $\sigma \circ D_2$ and $\sigma \circ loanReq \circ D_0$.

7 Related Work

There are various types of applications of using a model checker to generate tests. Ammann et al. combined model checking with *mutation analysis* to generate test cases [22]: after a specification model is mutated by applying mutation operators, a model checker generates counter-examples to distinguish the mutant models from the original specification model, and thus test cases are derived. Gargantini and Heitmeyer presented a technique to construct test sequences upon a special class of so-called *Software Cost Reduction* requirements, by using a model checker [23]. In order to save memory from a huge predefined test suite, Tretmans and de Vries [24] used model checker SPIN to generate tests *during testing* for non-deterministic stateful systems. How to generate test cases according to some *data flow test selection criteria* is discussed in [25]. In [20], Goltz et al. used a model checker to generate a shortest distinguishing sequence of an EFSM. In terms of applying model checking tools for test generation, we have added one more example along this line of research, particularly for testing in context.

Along the approaches of testing in context, there are several possible ways to interpret the context of an IUT. Petrenko et al. considered the situation where the IUT is an embedded component and its communication with the environment has to be carried out through its context. For this case, they presented

a framework of testing an embedded component in context [2, 16]. In particular, the problems of test executability and fault propagation are addressed in the presence of the context. In [15, 17–19], different approaches are discussed for solving the problem of translating internal tests derived for an embedded component into external observable tests of the entire system. Different from their test architecture, our work is applicable to testing an IUT that is associated with an embedded component.

8 Conclusion and Final Remarks

In this paper, we presented a method of deriving a *complete* test suite w.r.t. trace pre-order for testing the IUT with an embedded context, and provided a way of implementing this method by making use of model check tools.

As an initial piece of work on testing in context with model checkers, our focus has been put on the general method. Further improvements can be made in terms of the size of the constructed test suite. For example, it is possible to reduce the size of the generated test suite by constructing a *test tree* similar to the one introduced in [3]; We can adopt those model checkers that can always find *shortest* counter-examples in terms of the lengths so that shorter test sequences can be derived. Apart from the optimization issue, there are many other directions to extend our current work.

- It remains interesting to discuss our test generation technique in more general situations where both the IUT and its context have communications with the environment.
- IUT may be nondeterministic: we would like to study how to extend our results to nondeterministic testing in context.
- When the IUT is completely specified, it is not always possible to achieve trace equivalence due to the interoperability of the IUT and its context. We would like to discuss the condition on $\mathcal{S}$ and $\mathcal{C}$ such that trace equivalence can be achieved.
- We have used distinguishing sequence for state identification. At expense of its convenience for testing, distinguishing sequence does not always exist. Although the use of characterization set usually results in much bigger test suites, a characterization set is more likely to exist in an FSM with context. Therefore, we would like to study on how to use model checking tools to generate characterization set in our setting.

Acknowledgments. This work is supported by the Natural Sciences and Engineering Research Council of Canada under grant number RGPIN 209774.

References

1. Tretmans, J.: Conformance testing with labelled transition systems: Implementation relation and test generation. Computer Networks and ISDN Systems 29, 49–79 (1996)

2. Petrenko, A., Yevtushenko, N., von Bochmann, G., Dssouli, R.: Testing in context: framework and test derivation. Computer Communications 19(14), 1236–1249 (1996)
3. Chow, T.: Testing software design modeled by finite-state machines. IEEE Trans. Software Eng. SE-4(3), 178–187 (1978)
4. Sabnani, K., Dahbura, A.: A protocol test generation procedure. Computer Networks and ISDN Systems 4(15), 285–297 (1988)
5. Hennie, F.: Fault detecting experiments for sequential circuits. In: Proc. of 5th Ann. Symp. Switching Circuit Theory and Logical Design, pp. 95–110 (1964)
6. van der Bijl, M., Rensink, A., Tretmans, J.: Compositional testing with ioco. In: Petrenko, A., Ulrich, A. (eds.) FATES 2003. LNCS, vol. 2931, pp. 86–100. Springer, Heidelberg (2004)
7. Hierons, R.M., Ural, H.: UIO sequence based checking sequences for distributed test architectures. Information and Software Technology 45(12), 793–803 (2003)
8. Chen, J., Hierons, R.M., Ural, H., Yenigun, H.: Eliminating redundant tests in a checking sequence. In: Khendek, F., Dssouli, R. (eds.) TestCom 2005. LNCS, vol. 3502, pp. 146–158. Springer, Heidelberg (2005)
9. Gonenc, G.: A method for the design of fault detection experiments. IEEE Trans. Computers 19(6), 551–558 (1970)
10. Ural, H., Wu, X., Zhang, F.: On minimizing the lengths of checking sequences. IEEE Transactions on Computers 46(1), 93–99 (1997)
11. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines — a survey. Proceedings of The IEEE 84(8), 1090–1123 (1996)
12. Holzmann, G.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
14. Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In: Proc. of the DIMACS/SYCON workshop on Hybrid systems III: verification and control: verification and control, pp. 232–243 (1995)
15. Lima, L.P., Cavalli, A.R.: A progmatic approach to generating test sequences for embedded systems. In: Proc. of 10th International Workshop on Testing of Communicating Systems, pp. 125–140 (1997)
16. Petrenko, A., Yevtushenko, N., von Bochmann, G.: Fault models for testing in context. In: Proc. of Internation Conference on Formal Techniques for Networked and Distributed Systems, pp. 125–140 (1996)
17. Petrenko, A., Yevtushenko, N.: Testing faults in embedded components. In: Proc. of 10th International Workshop on Testing of Communicating Systems, pp. 272–287 (1997)
18. El-Fakih, K., Petrenko, A., Yevtushenko, N.: FSM test translation through context. In: Uyar, M.Ü., Duale, A.Y., Fecko, M.A. (eds.) TestCom 2006. LNCS, vol. 3964, pp. 245–258. Springer, Heidelberg (2006)
19. El-Fakih, K., Yevtushenko, N.: Fault propagation by equation solving. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 185–198. Springer, Heidelberg (2004)
20. Robinson-Mallett, C., Liggesmeyer, P., Mcke, T., Goltz, U.: Generating optimal distinguishing sequences with a model checker. ACM SIGSOFT Software Engineering Notes 30(4), 1–7 (2005)
21. Holzmann, G.: The Design and Validation of Computer Protocols. Prentice-Hall, Englewood Cliffs (1991)

22. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate test from specifications. In: Proc. of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM 1998), pp. 46–54 (1998)
23. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. ACM SIGSOFT Software Engineering Notes 24(6), 146–162 (1999)
24. de Vries, R., Tretmans, J.: On-the-fly conformance testing using Spin. International Journal on Software Tools for Technology Transfer 2(4), 382–393 (2000)
25. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: Proc. of IEEE ICSE 2003, pp. 232–242 (2003)

Requirements Coverage as an Adequacy Measure for Conformance Testing*

Ajitha Rajan[1], Michael Whalen[2], Matt Staats[1], and Mats P.E. Heimdahl[1]

[1] University of Minnesota
arajan@cs.umn.edu, staats@cs.umn.edu, heimdahl@cs.umn.edu
[2] Rockwell Collins Inc.
mwwhalen@rockwellcollins.com

Abstract. Conformance testing in model-based development refers to the testing activity that verifies whether the code generated (manually or automatically) from the model is behaviorally equivalent to the model. Presently the adequacy of conformance testing is inferred by measuring structural coverage achieved over the model. We hypothesize that adequacy metrics for conformance testing should consider *structural coverage over the requirements* either in place of or in addition to structural coverage over the model. Measuring structural coverage over the requirements gives a notion of how well the conformance tests exercise the required behavior of the system.

We conducted an experiment to investigate the hypothesis stating structural coverage over formal requirements is more effective than structural coverage over the model as an adequacy measure for conformance testing. We found that the hypothesis was rejected at 5% statistical significance on three of the four case examples in our experiment. Nevertheless, we found that the tests providing requirements coverage found several faults that remained undetected by tests providing model coverage. We thus formed a second hypothesis stating that complementing model coverage with requirements coverage will prove more effective as an adequacy measure than solely using model coverage for conformance testing. In our experiment, we found test suites providing both requirements coverage and model coverage to be more effective at finding faults than test suites providing model coverage alone, at 5% statistical significance. Based on our results, we believe existing adequacy measures for conformance testing that only consider model coverage can be strengthened by combining them with rigorous requirements coverage metrics.

1 Introduction

In critical avionics applications, the validation and verification phase (V&V) is particularly costly and consumes a disproportionably large share of the development resources. Thus, if the process of deriving test cases for V&V can be

* This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 86–104, 2008.

automated to provide test suites that satisfy the most stringent standards (such as DO-178B in civil avionics [20]), dramatic time and cost savings can be realized. The current trend towards model-based development is one attempt to address this problem. In model-based software development, the traditional testing process is split into two distinct activities: one activity that tests the model to *validate* that it accurately captures the customers' high-level requirements, and another testing activity that *verifies* whether the code generated (manually or automatically) from the model is behaviorally equivalent to (or conforms to) the model. (Note that by "model", we are referring specifically to a high level formal model written in a language such as Simulink or Lustre. Throughout this paper, we refer to this simply as a "model".) In this paper, we focus on the second testing activity—verification through conformance testing. There are currently several tools, such as model checkers, that provide the capability to automatically generate conformance tests [19, 7] from formal models. In this paper, we examine the effectiveness of metrics used in measuring the adequacy of the generated conformance tests.

For critical avionics software, DO-178B necessitates test cases used in verification to achieve requirements coverage in addition to structural coverage over the code. However, there is no direct and objective measure of requirements coverage, and adequacy of tests is instead inferred by examining structural coverage achieved over the model. The Modified Condition and Decision Coverage (MC/DC) used when testing highly critical software [20] in the avionics industry has been a natural choice to measure structural coverage for the most critical models. In our work [21], however, we have defined coverage metrics that provide *direct and objective* measures of how well a test suite exercises a set of high-level formal software requirements. We examined using requirements coverage metrics, in particular the Unique First Cause (UFC) coverage metric, to measure adequacy of tests used in model validation (or black-box testing) and found them to be useful. To save time and effort, we would like to re-use validation tests providing requirements coverage for verification of code through conformance testing as well. This paper examines the suitability of using tests providing requirements UFC coverage for conformance testing as opposed to tests providing MC/DC over the model.

We believe requirements coverage will be useful as an adequacy measure for conformance testing for several reasons. First, measuring structural coverage over the requirements gives a direct assessment of how well the conformance tests exercise the required behavior of the system. Second, if a model is missing functionality, measuring structural coverage over the model will not expose such defects of omission. Third, obligations for requirements coverage describe satisfying scenarios (paths) in the model as opposed to satisfying states defined by common model coverage obligations (such as MC/DC). We believe coverage obligations that define satisfying paths will necessitate longer and more effective test cases than those defining satisfying states in the model. Finally, we found in [16] that structural coverage metrics over the model, in particular MC/DC, are sensitive to the structure of the model used in coverage measurement. Therefore,

these metrics can be easily rendered inefficient by (purposely or inadvertently) restructuring the model to make it easier to achieve the desired coverage.

For these reasons, we believe that requirements coverage will serve as a stronger adequacy measure than model coverage in measuring adequacy of conformance test suites. More specifically, we investigate the following hypothesis in this paper:

Hypothesis 1 (H_1). *Conformance tests providing requirements UFC coverage are more effective at fault finding than conformance tests providing MC/DC over the model.*

We evaluated this hypothesis on four industrial examples from the civil avionics domain. The requirements for these systems are formalized as Linear Temporal Logic (LTL) [5] properties. The systems were modeled in the Simulink notation [12]. Using the Simulink models, we created implementations that we used as the basis for the generation of large sets of mutants by randomly seeding faults. We generate numerous test suites to provide 100% achievable UFC coverage over the LTL properties (the formal requirements), and numerous test suites to provide 100% achievable MC/DC over the model. We assessed the effectiveness of the different test suites by measuring their fault finding capability, i.e., running them over the sets of mutants and measuring the number of faults detected.

In our experiment we found that Hypothesis 1 was rejected on three of the four examples at the 5% statistical significance level. This result was somewhat disappointing since we believed that the requirements coverage would be effective as a conformance testing measure. The astute reader might point out that the result might not be surprising since the effectiveness of the requirements-based tests providing UFC coverage heavily depends on the 'goodness' of the requirements set; in other words, a poor set of requirements leads to poor tests. In this case, however, we worked with case examples with very good sets of requirements and we had expected better results. Nevertheless, we found that the tests providing requirements UFC coverage found several faults that remained undetected by tests providing MC/DC over the model. We thus formed a second hypothesis stating that complementing model coverage with requirements coverage will prove more effective as an adequacy measure than solely using model coverage for conformance testing. To investigate this, we formulated and tested the following hypothesis:

Hypothesis 2 (H_2). *Conformance tests providing requirements UFC coverage in addition to MC/DC over the model are more effective at fault finding than conformance tests providing only MC/DC over the model.*

In our second set of experiments, the combined test suites were significantly more effective than MC/DC test suites on three of the four case examples (at the 5% statistical significance level). For these examples, UFC suites found several faults not revealed by the MC/DC suites making the combination of UFC and MC/DC more effective than MC/DC alone. The relative improvement was in the range of $4.3\% - 10.8\%$ on these examples. We strongly believe that for

the case example that did not support Hypothesis 2, the MC/DC suite found all possible faults, making improvement with the combined suites impossible. Based on our results, we believe that existing adequacy measures for conformance testing based solely on structural coverage over the model (such as MC/DC) can be strengthened by combining them with requirements coverage metrics such as UFC. It is worth noting that Briand et.al. found similar results in their study [3], though in the context of state-based testing for complex component models in object-oriented software. Combining a state-based testing technique for classes or class clusters modeled with statecharts [8], with a black-box testing technique, category partition testing, proved significantly more effective in fault detection. We recommend future measures of conformance testing adequacy to consider both requirements and model coverage either by combining existing metrics, such as MC/DC and UFC, or by defining new metrics that account for both.

The remainder of the paper is organized as follows. Section 2 introduces our experimental setup and the case examples used in our investigation. Results and statistical analysis are presented in Section 3. Finally in Sections 4 and 5, we analyze and discuss the implications of our results, and point to future directions.

2 Experiment

We use four industrial systems in our experiment: two models from a display window manager for an air-transport class aircraft (DWM_1, DWM_2), and two models representing flight guidance mode logic for business and regional jet class aircrafts (Vertmax_Batch and Latctl_Batch). All four systems were viewed to have good sets of requirements as judged by the developer of the system. We conducted the experiments for each case example using the steps outlined below (elaborated in later sections):

1. Generate and reduce test suites to provide requirements UFC coverage: We generated a test suite to provide UFC coverage over the formalized LTL requirements. This test suite was naïvely generated, one test case for every UFC obligation, and thus highly redundant. We reduced the test suite randomly while maintaining UFC coverage over the requirements. We generated three such randomly reduced test suites.

2. Generate and reduce test suites to provide MC/DC over the model: We naïvely generated a test suite to provide MC/DC over the model. We then randomly reduced the test suite to maintain MC/DC over the model. We generated three such reduced test suites.

3. Combined test suites that provide MC/DC + requirements UFC: Among the reduced MC/DC suites from the previous step, we selected the most effective MC/DC test suite based on their fault finding ability. We merge this test suite with each of the reduced UFC test suites from the first step. The combined suites thus provide both MC/DC over the model and UFC coverage over the requirements.

4. Generate mutants: We randomly seeded faults in the correct implementation and generated three sets of 200 mutants using the method outlined in Section 2.3.

5. Assess and compare fault finding: We run each of the test suites from steps 1, 2 and 3 (that provide requirements UFC coverage, MC/DC over the model, and MC/DC + requirements UFC coverage respectively) against each set of mutants and the model. Note that the model serves as the oracle implementation in conformance testing. We say that a mutant is killed (or detected) by a test suite if any of the test cases in the suite results in different output values between the model and the mutant. We recorded the number of mutants killed by each test suite and computed the fault finding ability as the percentage of mutants killed to the total number of mutants seeded.

2.1 Case Examples

In our experiment, we use four industrial systems. All four systems were modeled using the Simulink notation from Mathworks Inc.

Display Window Manager Models (DWM_1 and DWM_2): The Display Window Manager models, DWM_1, and DWM_2, represent 2 of the 5 major subsystems of the Display Window Manager (DWM) of an air transport-level commercial displays system. The DWM acts as a 'switchboard' for the system and has several responsibilities related to routing information to the displays and manages the location of two cursors that can be used to control applications by the pilot and copilot.

Flight Guidance System: A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. The two FGS models in this paper focus on the mode logic of the FGS. The Vertmax_Batch and Latctl_Batch models describe the vertical and lateral mode logic for the flight guidance system.

2.2 Test Suite Generation and Reduction

We generated test suites to provide UFC coverage over formal LTL requirements and to provide MC/DC over the model. The approach to generate and reduce the test suites for the two different coverage measures is detailed below. Additionally, we merge the reduced test suites for the two coverage measures to create combined test suites that provide UFC coverage over the requirements in addition to MC/DC over the model.

UFC Coverage over Requirements: The requirements coverage metric used in this paper is the *Unique First Cause* (UFC) coverage defined in [21].

The UFC metric is adapted from the Modified Condition/Decision Coverage (MC/DC) criterion [4, 9] defined over source code that defines satisfying states in the implementation. Since requirements captured as LTL properties define paths rather than states, we broadened our view of structural coverage to accommodate satisfying paths rather than satisfying states. We defined these satisfying test paths by extending the constraints for state-based MC/DC to include temporal operators. A test suite is said to satisfy UFC coverage over a set of LTL formulae if executing the test cases in the test suite will guarantee that:

- Every basic condition in a formula has taken on all possible outcomes at least once
- Each basic condition has been shown to independently affect the formula's outcome.

We defined independence in terms of the shortest satisfying path for the formula. Thus, if we have a formula A and a path π, an atomic condition α in A is the unique first cause if, in the first state along π in which A is satisfied, it is satisfied because of atomic condition α. The formal definition for UFC and the obligations for LTL temporal operators was presented in [21].

The notion of requirements UFC coverage used in this paper is related to work assessing the completeness and correctness of formulae in temporal logics, in particular, *vacuity checking* of temporal logic formulas [2, 10, 15]. The focus of this paper, however, is on the application and usefulness of requirements coverage to measure adequacy of conformance testing, and not on the completeness of requirements. We are not aware of studies that investigated applicability of requirements coverage in this context.

Several research efforts have developed techniques for automatic generation of tests from formal models using model checkers as test case generation tools [17, 18, 7]. One such technique operates by formulating a test criterion as a verification condition for the model checker. The obligations for requirements UFC coverage are given as trap properties (by negating the obligations) to the model checker along with the formal model of the system, and the model checker returns counter examples that constitute a test suite for UFC coverage over the LTL requirements.

A test suite thus generated will be highly redundant, as a single test case will often satisfy several UFC obligations. We therefore reduce this test suite using a greedy approach. We randomly select a test case from the test suite, check how many UFC obligations are satisfied and add it to a reduced test set. Next, we randomly pick another test case from the suite and check whether any additional UFC obligations were satisfied. If so, we add the test case to the reduced test set. This process continues till we have exhausted all the test cases in the test suite. We now have a randomly reduced test suite that maintains UFC coverage over the LTL requirements. We generate three such reduced UFC test suites for each case example in our experiment to eliminate the possibility of skewing our results with an outlier (an extremely good or bad reduced test suite).

MC/DC over model: The full test suite to provide MC/DC used in this experiment is the same one used in previous work [16]. We used the test suite that provides MC/DC over the inlined model rather than the non-inlined model as it is more rigorous and effective. We thus compare the requirements UFC coverage against a rigorous notion of MC/DC. The test suite was automatically generated using the NuSMV [13] model checker to provide MC/DC over the model. The full test suite was naïvely generated, with a separate test case for every construct we need to cover in the model. This straightforward method of generation results in highly redundant test suites, as with UFC test suite generation. Thus, the size of the complete test suite can typically be reduced while preserving coverage.

The approach to reduce the test suite is similar to that used for UFC coverage. As before, we generate three such reduced test suites to decrease the chances of skewing our results with an outlier (very good or very bad reduced test suite).

Requirements UFC Coverage + MC/DC over model: To generate test suites providing both requirements UFC coverage and MC/DC over the model, we simply merge the test suite providing UFC with the test suite providing MC/DC. As mentioned previously, we generated three reduced MC/DC suites and three reduced UFC suites. It is thus possible to create nine different combined suites, by merging each of the three reduced MC/DC suites with each of the three reduced UFC suites. Using all nine suites in our experiment, however, would have been very time consuming. To reduce the combinations, we instead elected to use only the best reduced MC/DC suite (with respect to fault finding among the reduced MC/DC suites) for creating the combined test suites. We thus merge the best MC/DC suite with each of the three reduced UFC suites to create only three combined suites. Note that choosing the best MC/DC implies that the combined suites must improve the fault finding of the best MC/DC suite to support Hypothesis 2.

2.3 Mutant Generation

To create mutants or faulty implementations, we built a fault seeding tool that can randomly inject faults into the implementation. Each mutant is created by introducing a single fault into a correct implementation by mutating an operator or variable.

The fault seeding tool is capable of seeding faults from different classes. We seeded the following classes of faults:

Arithmetic: Changes an arithmetic operator (+, -, /, *, mod, exp).
Relational: Changes a relational operator ($=, \neq, <, >, \leq, \geq$).
Boolean: Changes a boolean operator ($\vee, \wedge$, XOR).
Negation: Introduces the boolean $\neg$ operator.
Delay: Introduces the delay operator on a variable reference (that is, use the stored value of the variable from the previous computational cycle rather than the newly computed value).

Constant: Changes a constant expression by adding or subtracting 1 from int and real constants, or by negating boolean constants.

Variable Replacement: Substitutes a variable occurring in an equation with another variable of the same type.

To seed a fault from a certain class, the tool first randomly picks one expression among all possible expressions of that kind in the implementation. It then randomly determines how to change the operator. For instance to seed an arithmetic mutation, we first randomly pick one expression from all possible arithmetic expressions to mutate, say we pick the expression 'a + b'; we then randomly determine if the arithmetic operator '+' should be replaced with '-' or '*' or '/' and create the arithmetic mutant accordingly. Our fault seeding tool ensures that no duplicate faults are seeded.

In our experiment, we generated mutants so that the 'fault ratio' for each fault class is uniform. The term fault ratio refers to the number of mutants generated for a specific fault class versus the total number of mutants possible for that fault class. For example, assume an implementation consists of R Relational operators and B Boolean operators. Thus there are R possible Relational faults and B possible Boolean faults. For uniform fault ratio, we would seed x relational faults and y boolean faults in the implementation so that $x/R = y/B$.

We generated three sets of 200 mutants for each case example. We generated multiple mutant sets for each example to reduce potential bias in our results from a mutant set that may have very hard (or easy) faults to detect. Our mutant generator does not guarantee that a mutant will be semantically different from the original implementation. Nevertheless, this weakness in mutant generation does not affect our results, since we are investigating the relative fault finding of test suites rather than the absolute fault finding.

The fault finding effectiveness of a test suite is measured as the number of mutants detected (or 'killed') to the total number of mutants created. We say that a mutant is detected by a test suite when the test suite results in different observed values between the mutant and the oracle implementation. The system model serves as the oracle implementation in conformance testing. We only observe the output values of the model and mutants for comparison. We do not use internal state information, as internal state information between the model and implementation may differ and can therefore not be compared directly. Additionally, internal state information of the system under test may not be available during real-world tests and it is therefore preferable to perform the comparison with only output values.

3 Experimental Results

For each case example described in Section 2.1, we generated three reduced UFC test suites, three reduced MC/DC test suites, three combined UFC + MC/DC test suites and three sets of mutants. As mentioned earlier the combined suites are created by merging the best reduced MC/DC suite with each of the three reduced UFC suites. For this reason we compare the fault finding ability of the

combined suites only against the best MC/DC suite rather than all the reduced MC/DC suites. We ran every test suite against every set of mutants, and recorded the percentage of mutants caught. For each case example, this yielded nine observations each for MC/DC, UFC and the combined test suites. We average the percentage of mutants caught across the mutant sets for each case example and each kind of test suite. This yields three averages, one each for MC/DC, UFC, and combined test suites as summarized in Table 1. Also, for each case example we identify the most effective MC/DC suite (among the generated three reduced MC/DC suites) and calculate average fault finding across the mutant sets. The 'Best MC/DC' column in the table represents these averaged observations. Table 1 also gives the relative improvement in average fault finding of UFC suites over MC/DC test suites, and combined suites over the best MC/DC suite. Note that some of the numbers in the relative improvement column in Table 1 are negative. This implies that the test suite did not yield an improvement, and instead did worse than the MC/DC test suite at fault finding. For instance, for the DWM_1 model the MC/DC test suites provide an average fault finding of 84.6% and the UFC suites provide an average fault finding of 82.7%, and thus the relative improvement in fault finding for UFC suites is negative (= -2.2%) with respect to MC/DC suites. Conversely, for the DWM_1 system, the combined suites provide better fault finding (an average of 91.5%) than the best MC/DC suite (85.8%), giving a positive relative improvement of 6.6%.

The complete set of 27 fault finding observations for each case example is presented in Table 2. Note that in the table, $M1$, $M2$, $M3$ denote the three mutant sets; $MCDC\_1$, $MCDC\_2$, $MCDC\_3$ refer to the reduced MC/DC suites; UFC_1, UFC_2, UFC_3 to reduced UFC suites; and C_1, C_2, C_3 to the combined UFC and MC/DC suites. The best MC/DC suite (used to create combined suites) can be identified by comparing the fault finding of $MCDC\_1$, $MCDC\_2$, and $MCDC\_3$ across the mutant sets. Thus from the results in Table 2 we find for the DWM_1 system, the best MC/DC test suite is $MCDC\_3$. For DWM_2 and Vertmax_Batch systems, all three reduced MC/DC suites are equally effective so any of them can be used for creating the combined suites. We randomly selected $MCDC\_1$ for DWM_2 and $MCDC\_3$ for Vertmax_Batch system. Finally, for the Latctl_Batch system, $MCDC\_1$ is the most effective.

From the results in Tables 1 and 2, it is evident that for all case examples, except the Latctl_Batch system, MC/DC test suites outperform the UFC suites

Table 1. Average percentage of mutants caught by test suites and relative improvement over MC/DC

	Avg. MC/DC	**Avg. UFC**	**Relative Improv.**	**Best MC/DC**	**Avg. Combined**	**Relative Improv.**
DWM_1	84.6%	82.7%	-2.2%	85.8%	91.5%	6.6 %
DWM_2	90.6%	16.7%	-81.6%	90.6%	90.6%	0.0%
Latctl_Batch	85.1%	88.7%	4.2%	85.4%	94.6%	10.8%
Vertmax_Batch	86.0%	68.6%	-20.2%	86.0%	89.7%	4.3%

Table 2. Complete results for all case examples

	DWM_1								
	MCDC_1	**MCDC_2**	**MCDC_3**	**UFC_1**	**UFC_2**	**UFC_3**	**C_1**	**C_2**	**C_3**
M1	82.7%	81.2%	84.3%	79.2%	79.7%	76.1%	88.3%	89.3%	88.8%
M2	83.8%	83.8%	86.3%	83.8%	82.7%	81.2%	91.4%	91.9%	91.4%
M3	86.3%	86.3%	86.8%	87.3%	87.8%	86.8%	93.9%	94.4%	94.4%
TS Size	73	76	77	463	469	468	540	546	545
	DWM_2								
	MCDC_1	**MCDC_2**	**MCDC_3**	**UFC_1**	**UFC_2**	**UFC_3**	**C_1**	**C_2**	**C_3**
M1	91.4%	91.4%	91.4%	17.2%	15.2%	16.7%	91.4%	91.4%	91.4%
M2	91.4%	91.4%	91.4%	16.7%	14.6%	16.7%	91.4%	91.4%	91.4%
M3	88.9%	88.9%	88.9%	18.7%	16.2%	18.7%	88.9%	88.9%	88.9%
TS Size	452	452	448	33	32	31	485	484	483
	Latctl_Batch								
	MCDC_1	**MCDC_2**	**MCDC_3**	**UFC_1**	**UFC_2**	**UFC_3**	**C_1**	**C_2**	**C_3**
M1	85.2%	84.2%	84.7%	89.3%	87.8%	89.8%	94.4%	92.9%	91.8%
M2	85.7%	85.7%	85.2%	89.3%	89.3%	89.8%	96.4%	95.9%	95.9%
M3	85.2%	84.7%	85.2%	88.8%	85.2%	88.8%	94.9%	93.9%	94.9%
TS Size	73	71	73	50	49	53	123	122	126
	Vertmax_Batch								
	MCDC_1	**MCDC_2**	**MCDC_3**	**UFC_1**	**UFC_2**	**UFC_3**	**C_1**	**C_2**	**C_3**
M1	83.8%	83.8%	83.8%	67.5%	69.5%	66.0%	88.8%	88.8%	88.8%
M2	81.3%	81.3%	81.3%	71.1%	71.1%	71.6%	91.9%	90.4%	90.4%
M3	88.9%	88.9%	88.9%	64.5%	66.0%	70.1%	87.3%	86.3%	88.3%
TS Size	301	299	297	89	79	88	386	376	385

in fault finding. The degree to which MC/DC suites are better, however, varies by a vast range. The maximum difference is on DWM_2, where MC/DC suites provide an average fault finding of 90.6% in contrast to 16.7% provided by UFC suites. The minimum difference is on DWM_1 where MC/DC provides an average fault finding of 84.6% versus 82.7% provided by UFC suites. The combined suites on the other hand outperform the MC/DC suites. The relative improvement provided by the combined suites however spans a much smaller range (0 - 10.8%). In other words, the number of different faults revealed by the UFC suites as compared to the best MC/DC suite is in the range of $0 - 10.8\%$ of the mutants seeded. The combined suites provide better fault finding than the best MC/DC suite on three of the four case examples. On the DWM_2 system the combined suites yield no improvement. A detailed discussion of the implication of these results is presented in Section 4.

3.1 Statistical Analyses

In this section, we statistically analyze the results in Tables 1 and 2 to determine if the hypotheses H_1 and H_2, stated previously in Section 1, are supported.

To evaluate H_1 and H_2, we formulate our respective null hypotheses $H0_1$ and $H0_2$ as follows:

$H0_1$: A test suite generated to provide requirements UFC coverage will find the same number of faults as a test suite generated to provide MC/DC coverage over the model.

$H0_2$: A test suite generated to provide both requirements UFC coverage and MC/DC over the model will reveal the same number of faults as a test suite generated to provide only MC/DC over the model.

To accept H_2, we must reject $H0_2$. Rejecting $H0_2$ implies that the data for the combined test suite and MC/DC suite come from different populations. In other words, this implies that either the combined suites have more fault finding than the MC/DC suites or vice versa. However, the combined suite includes the MC/DC suite and can therefore never have lesser fault finding than the MC/DC suite. This implies that H_2 is supported when $H0_2$ is rejected. On the other hand, rejecting $H0_1$ does not necessarily imply H_1 is supported, as this implies that the UFC suites have *different* fault finding ability than the MC/DC suites, not necessarily better fault finding ability. To accept H_1 after rejecting $H0_1$, we examine the data in the table and determine if the UFC suites have greater fault finding than the MC/DC suite. If so, we accept H_1. If the data indicates that UFC suites instead have lesser fault finding than the MC/DC suites, we reject H_1.

Our experimental observations are drawn from an unknown distribution, and we therefore cannot reasonably fit our data to a theoretical probability distribution. To evaluate $H0_1$ and $H0_2$ without any assumptions on the distribution of our data, we use the permutation test, a non-parametric test with no distribution assumptions. When performing a permutation test, a reference distribution is obtained by calculating all possible permutations of the observations [6, 11]. To perform the permutation test, we restate our null hypotheses as:

$H0_1$: The data points for percentage of mutants caught using the UFC and MC/DC test suites come from the same population.

$H0_2$: The data points for percentage of mutants caught using the MC/DC and combined UFC + MC/DC test suites come from the same population.

We evaluate the two hypotheses for each of the case examples. The procedure for permutation test of each hypothesis is as follows. Data is partitioned into two groups: A and B. Null hypothesis states that data in groups A and B come from the same population. We calculate the test statistic S as the absolute value of the difference in the means of group A and B:

$$S = \mid \overline{A} - \overline{B} \mid$$

We calculate $Number\ of\ Permutations$ as the number of ways of grouping all the observations in A and B into two sets. We then let $COUNT$ equal the number of permutations of A and B in which the test statistic is greater than S. Finally, $P - Value$ is calculated as:

$$P - Value = COUNT\ /\ Number\ of\ Permutations$$

For each case example, if $P - Value$ is less than the α value of 0.05 then we reject the null hypothesis with significance level α.

The null hypotheses $H0_1$ and $H0_2$ are evaluated using different groups of data. For $H0_1$, data for each case example in Table 2 is partitioned into two groups with nine observations each: % of faults caught by UFC test suites (group A – columns UFC_1, UFC_2, UFC_3 in the table), and % of faults caught by MC/DC test suites (group B – columns $MCDC\_1$, $MCDC\_2$, $MCDC\_3$). We calculate the $Number\ of\ Permutations$ as:

$$Number\ of\ Permutations = \binom{18}{9} = 48620$$

For $H0_2$, data for each case example in Table 2 is partitioned into two groups, one with nine observations and the other with three observations: % of faults caught by combined UFC+MC/DC test suites (group A – columns C_1, C_2, C_3), and % of faults caught by the best MC/DC suite (group B – $MCDC\_1$ column for DWM_2 and Latctl_Batch systems, and $MCDC\_3$ column for DWM_1 and Vertmax_Batch systems). We calculate the $Number\ of\ Permutations$ as:

$$Number\ of\ Permutations = \binom{12}{9} = 220$$

We then determine the p-value for each hypothesis using the procedure described previously. Table 3 lists the p-values for both null hypotheses ($H0_1$ and $H0_2$) and states if the corresponding original hypotheses (H_1 and H_2) are supported for each case example. As mentioned earlier, for each case example, H_1 is supported if $H0_1$ is rejected with significance level $\alpha = 0.05$ and all the UFC suites (columns UFC_1, UFC_2, UFC_3 in Table 2) have better fault finding than the MCDC suites (columns $MCDC\_1$, $MCDC\_2$, $MCDC\_3$), and H_2 is supported if we reject $H0_2$.

Given the p-values in Table 3 and the fault finding data in Table 2 we examine why the original hypotheses (H_1 and H_2) are supported/rejected for each case example. For the DWM_1 system, $H0_1$ is accepted (since p-value is greater than α value), and we therefore reject H_1. For the other three systems, $H0_1$ is rejected but the UFC suites outperform the MC/DC suites only on the Latctl_Batch system. For the DWM_2 and Vertmax_Batch systems, MC/DC suites always outperform the UFC test suites. Thus, H_1 is supported on the Latctl_Batch system and rejected on the DWM_2 and Vertmax_Batch systems. On the other hand, $H0_2$ is rejected (p-value less than the α value) on all but the DWM_2 system. This implies that H_2 is supported on all except the DWM_2 system. Thus, we find that

Table 3. Hypotheses Evaluation for different case examples

	P-Value		**Result**	
	$H0_1$	$H0_2$	H_1	H_2
DWM_1	0.24	0.004	Unsupported	Supported
DWM_2	0.00004	1.0	Unsupported	Unsupported
Latctl_Batch	0.0002	0.004	Supported	Supported
Vertmax_Batch	0.00004	0.027	Unsupported	Supported

with statistical significance level $\alpha = 0.05$ hypothesis H_1 is supported only on one case example, and hypothesis H_2 is supported on three of the four case examples.

3.2 Threats to Validity

While our results are statistically significant, they are derived from a small set of examples, which poses a threat to the generalization of the results. Nevertheless, we believe that the examples in our experiment are highly representative and our results are generalizable to systems within the same domain.

Our fault seeding method seeds one fault per mutant. In practice, implementations are likely to have more than one fault. However, previous studies have shown that mutation testing in which one fault is seeded per mutant draws valid conclusions of fault finding ability [1].

Additionally, all fault seeding methods have an inherent weakness. It is difficult to determine the exact fault classes and ensure that seeded faults are representative of faults that occur in practical situations. In our experiment, we assume a uniform ratio of faults across fault classes. This may not reflect the fault distribution in practice. Finally, our fault seeding method does not ensure that seeded faults result in mutants that are semantically different from the oracle implementation. Ideally, we would eliminate mutants that are semantically equivalent, however, identifying such mutants is infeasible in practice.

4 Discussion

In this section we analyze and discuss the implications of the results in Tables 1 and 2. We present the discussion in the context of Hypotheses 1 and 2 stated in Section 1.

4.1 Analysis - Hypothesis 1

As seen from Table 1, on all but one of the industrial systems, test suites generated for requirements UFC coverage have lower fault finding than test suites providing MC/DC over the system model. Statistical analysis revealed that hypothesis 1 stating "test suites providing requirements UFC coverage have better fault finding than test suites providing MC/DC over the model" was supported only on the Latctl_Batch system and rejected on all the other systems at the 5% significance level. We believe this may be because of one or both of the following reasons, (1) The UFC metric used for requirements coverage is not sufficiently rigorous and we thus have an inadequate set of requirements-based tests, and (2) Requirements are not sufficiently defined with respect to the system model. Thus, test suites providing requirements coverage will be ineffective at revealing faults in the model since there are behaviors in the model not specified in the requirements.

To assess the rigor of the UFC metric and the quality of the requirements with regard to behaviors covered in the system model, we measured MC/DC achieved by the reduced UFC suites over the system model. The results are summarized in Table 4. We found that for all the case examples, UFC test suites provide less

Table 4. MC/DC achieved by the reduced UFC suites over the system model

	Avg. MC/DC Achieved by UFC suites	Achievable MC/DC	Rel. Diff.
DWM_1	78.2%	92.5%	15.5%
DWM_2	25.8%	100%	74.2%
Latctl_Batch	88.6%	98.0%	9.6%
Vertmax_Batch	80.9%	99.8%	18.9%

than Achievable MC/DC over the system model. Thus, faults seeded in these uncovered portions of the model cannot be revealed by the UFC suites. The extent to which the model is covered is an indicator of the effectiveness of the UFC metric and the quality of the requirements set. On the DWM_1, Vertmax_Batch, and Latctl_Batch systems the UFC suites do reasonably well, achieving an average MC/DC of 78.2%, 88.6%, and 80.9% respectively as compared to 92.5%, 98% and 99.8% achievable MC/DC. Note, however, that relative differences in MC/DC need not correspond exactly to relative differences in fault finding between the UFC and MC/DC suites (as seen in our examples). In addition to coverage, fault finding is also highly influenced by the nature and number of faults seeded in covered and uncovered portions of the model. The relation between coverage and fault finding is not the focus of this paper and we hope to investigate this in our future work.

On the DWM_2 system, the UFC suites do poorly in both fault finding and MC/DC achieved. The UFC suites only achieve an average of 25.8% MC/DC over the model when compared to an achievable MC/DC of 100%. Correspondingly, the UFC suites have very poor fault finding (average of 16.7%) when compared to the MC/DC suites (average of 90.6%), since faults seeded in the uncovered portions of the model cannot be revealed by the UFC suites. The terrible fault finding and MC/DC achieved by the UFC suites on the DWM_2 system was surprising since we knew the system had a good set of requirements. To gain better understanding we took a closer look at the requirements set and the UFC obligations generated from them. We found that many of the requirements were structured similar to the sample requirement (formalized as an LTL property in the SMV [13] language) below,

```
LTLSPEC G(var_a > (
    case
        foo : 0 ;
        bar : 1 ;
    esac +
    case
        baz : 2 ;
        bpr : 3 ;
    esac
));
```

Informally, the sample requirement states that *var_a* is always greater than the sum of the outcomes of the two case expressions. When we perform UFC for the above requirement, it would result in obligations for the following expressions:

1. Relational expression within the globally operator (G)
2. Atomic condition foo within the first case expression
3. Atomic condition bar within the first case expression
4. Atomic condition baz within the second case expression
5. Atomic condition bpr within the second case expression

The above requirement may be restructured (to express the same behavior) so that the sum of two case expressions is expressed as a single case expression as shown:

```
LTLSPEC G(var_a > (
    case
        foo & baz : 0 + 2 ;
        foo & bpr : 0 + 3 ;
        bar & baz : 1 + 2 ;
        bar & bpr : 1 + 3 ;
    esac
));
```

Achieving UFC coverage over this restructured requirement will involve more obligations than before since the boolean conditions in the case expression are more complex. UFC would result in obligations for the following expressions in this restructured requirement:

1. Relational expression within the globally operator (G)
2. Complex condition foo & baz within the case expression
3. Complex condition foo & bpr within the case expression
4. Complex condition bar & baz within the case expression
5. Complex condition bar & bpr within the case expression

Thus, the structure of the requirements has a significant impact on the number and rigor of UFC obligations and hence the size of the test suite providing UFC coverage. In our experiment, we did not restructure requirements similar to the sample requirement discussed and instead retained the original structure. Therefore, the UFC obligations generated were fewer and far less rigorous. We believe this is the primary reason for the poor performance (both fault finding and MC/DC achieved) of the UFC suites for the DWM_2 system. The experience with the DWM_2 system suggests that even with a good set of requirements, rigorous requirements coverage metrics, such as the UFC metric, can be easily cheated since they are highly sensitive to the structure of the requirements. The issue here is similar to the sensitivity of the MC/DC metric to structure of the implementation observed in [16]. MC/DC was found to be significantly less effective when used over an implementation structure with intermediate variables and non inlined function calls as opposed to an implementation with inline expanded intermediate variables and function calls. Thus, as with all structural coverage metrics, we must be aware that the *structure* of the object used in measurement plays an important role in the effectiveness of the metrics.

To summarize, we find that the fault finding effectiveness of test suites providing requirements UFC coverage is heavily dependent on the nature and completeness of the requirements. Additionally, the rigor and robustness (with respect to requirements structure) of the requirements coverage metric used plays an important role in the effectiveness of the generated test suites. Thus, even with a good set of requirements, test suites providing requirements structural coverage may be ineffective if the coverage metric can be cheated. In our experiment, the UFC metric gets cheated when requirements are structured to hide the complexity of conditions on the DWM_2 system. Based on these observations and our results, we do not recommend using requirements coverage in place of model coverage as a measure of adequacy for conformance test suites.

4.2 Analysis - Hypothesis 2

As seen in Tables 1 and 2, for three of the four industrial case examples the combined UFC and MC/DC suites outperform the MC/DC suite in fault finding. For the DWM_2 system, however, the combined suites yield no improvement in fault finding over the MC/DC suite. Statistical analysis on the data in Table 2 revealed that Hypothesis 2 is supported with a significance level of 5% for the DWM_1, Vertmax_Batch, Latctl_Batch systems, and rejected for the DWM_2 system since the combined suites yield no improvement.

For the the DWM_1, Vertmax_Batch, Latctl_Batch systems, the combined UFC and MC/DC suites yielded an average fault finding improvement in the range of (4.3% - 10.8%) over the best MC/DC suite. The relative improvement implies that the UFC suites find a considerable number of faults not revealed by the best MC/DC suite.

To confirm that the improvement seen in DWM_1, Vertmax_Batch, Latctl_Batch systems is a result of combining the MC/DC metric with the UFC metric and not solely because of the increased number of test cases in the combined suites, we decided to measure the UFC coverage achieved over the requirements by the MC/DC suite. The results are summarized in Table 5. To understand the implications of the results in the table, consider the following two situations. If the MC/DC suite provides 100% achievable UFC over the requirements, it implies that the combined MC/DC + UFC coverage is satisfied by simply using the MC/DC suite instead of the combined suites. Under such circumstances, the fault finding improvement observed on combining the test suites would be solely due to the increased number of test cases. On the

Table 5. UFC achieved by the reduced MC/DC suites over the system model

	Avg. UFC Achieved by MC/DC suites	**Achievable UFC**	**Rel. Diff.**
DWM_1	28.3%	96.9%	70.8%
DWM_2	59.7%	64.0%	6.7%
Latctl_Batch	94.7%	99.5%	4.8%
Vertmax_Batch	97.4%	99.0%	1.6%

other hand, if the MC/DC suite provides less than achievable UFC over the requirements, it implies that there are scenarios/behaviors specified by the requirements that are not covered by the MC/DC suite but covered by the UFC suite. Thus, the combination may have proved more effective because of these additional covered scenarios and not simply because of increased test cases. We now take a closer look at the results in Table 5 to see which of these situations occurred. We found that in all three systems (Latctl_Batch, Vertmax_Batch, and DWM_1), the MC/DC suites provided less than achievable UFC coverage over the requirements. This indicates that the UFC suites cover several behaviors specified in the requirements that are not covered by the MC/DC suite. We postulate that these additional covered behaviors contribute to the improved fault finding observed with the combined suites on these systems.

For the DWM_2 system, the combined suites yield no improvement in fault finding over the MC/DC suite, implying that the faults revealed by the UFC suites are a subset of the faults revealed by the MC/DC suite. The DWM_2 system consists almost entirely of complex Boolean mode logic, and the MC/DC metric is extremely effective for these type of systems. There is thus a distinct possibility that the MC/DC suite reveals all the seeded faults (excluding semantically equivalent faults that can never be revealed). This belief was strengthened when we ran the full rather than reduced MC/DC and UFC suites and measured fault finding. We found that even with the full test suites, which have a dramatically larger number of test cases, the combination did not yield any improvement in the number of faults revealed. Therefore, we believe that there is a strong possibility the MC/DC suites revealed all but the semantically equivalent faults on the DWM_2 system. Under such circumstances, no test suite complementing the MC/DC suite can improve the fault finding, thus forcing us to always reject Hypothesis 2. Such occurrences are anomalous and we discount them from our analysis.

To summarize, we found that for three of the four case examples, the combined test suite providing both requirements UFC coverage and MC/DC over the model is significantly more effective than a test suite solely providing MC/DC over the model. For the DWM_2 system that did not support this, we strongly believe that the MC/DC suites revealed all possible faults making improvement in fault finding on combining with UFC suites impossible. We disregard this abnormal occurrence to conclude that combined test suites have better fault finding than the MC/DC suites for all the systems. Given our results, we believe using requirements coverage metrics, such as UFC, in combination with model coverage metrics, such as MC/DC, yields a significantly stronger adequacy measure than simply covering the model.

Note that for all the case examples, all three kinds of test suites—MC/DC, UFC, and combined—never yield 100% fault finding. This is because some of the seeded faults may result in mutant implementations that are semantically equivalent to the correct implementation (i.e., faults that *cannot* result in any observable failure). This is a common problem in fault seeding experiments [1, 14]. In industrial size examples it is extraordinarily expensive and time consuming, or—in most cases—infeasible to identify mutations that are semantically

equivalent to the correct implementation and exclude them from consideration. Therefore, the fault finding percentage that we give in our experiment results is a conservative estimate, and we expect the actual fault finding for the test suites to be higher if we were to exclude the semantically equivalent mutations. However, this issue will not affect our conclusions since we only judge based on relative fault finding rather than absolute fault finding.

5 Conclusions

Presently in model-based development, adequacy of conformance test suites is inferred by measuring structural coverage achieved over the model. In this paper we investigated the use of requirements coverage as an adequacy measure for conformance testing. Our empirical study revealed that on three of the four industrial case examples, our hypothesis stating "Requirements coverage (UFC) is more effective than model coverage (MC/DC) when used as an adequacy measure for conformance test suites" was rejected at 5% statistical significance level. Nevertheless, we found that requirements coverage is useful when used in combination with model coverage to measure adequacy of conformance test suites. Our hypothesis stating that "test suites providing both requirements UFC coverage and MC/DC over the model are more effective than test suites providing only MC/DC over the model" was supported at 5% significance level on three of the four case examples. The relative improvement yielded by the combined suites over the MC/DC suites was in the range of $4.3\% - 10.8\%$. The system that did not support the hypothesis was an outlier where we firmly believe the MC/DC suite found all possible faults, making improvement with the combined suites impossible. Based on our results, we believe that the effectiveness of adequacy measures based solely on model coverage can certainly be improved. Combining existing metrics for model coverage and requirements coverage investigated in this paper may be one possible way of accomplishing this. There may be other approaches, for instance, defining a new metric that accounts for both requirements and model coverage. We hope to investigate this further in our future work.

Another observation gained in our experiment relates to the sensitivity of requirements coverage metrics such as UFC to the structure of the requirements. Test suites providing requirements coverage may be ineffective even with an excellent set of requirements. This can occur when structure of the formalized requirements effectively "cheats" the requirements coverage metric. The UFC metric in our experiment was cheated when requirements were structured to hide the complexity of conditions in them. In our future work, we hope to define requirements coverage metrics that are more robust to the structure of the requirements.

References

1. Andrews, J.H., Briand, L.C., Labiche, Y.: Is Mutation an Appropriate Tool for Testing Experiments? In: Proceedings of the 27th International Conference on Software Engineering (ICSE), pp. 402–411 (2005)

2. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in ACTL formulas. In: Formal Methods in System Design, pp. 141–162 (2001)
3. Briand, L.C., Di Penta, M., Labiche, Y.: Assessing and Improving State-Based Class Testing: A Series of Experiments. IEEE Transactions on Software Engineering 30(11) (2004)
4. Chilenski, J.J., Miller, S.P.: Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal, 193–200 (September 1994)
5. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
6. Fisher, R.A.: The Design of Experiment. Hafner, New York (1935)
7. Gargantini, A., Heitmeyer, C.: Using model checking to generate tests from requirements specifications. Software Engineering Notes 24(6), 146–162 (1999)
8. Harel, D., Marelly, R.: Come Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer, New York (2003)
9. Hayhurst, K.J., Veerhusen, D.S., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. Technical Report TM-2001-210876, NASA (2001)
10. Kupferman, O., Vardi, M.Y.: Vacuity detection in temporal model checking. Journal on Software Tools for Technology Transfer 4(2) (February 2003)
11. Kvam, P.H., Vidakovic, B.: Nonparametric Statistics with Applications to Science and Engineering (2007)
12. Mathworks Inc. Simulink product web site, `http://www.mathworks.com/products/simulink`
13. The NuSMV Toolset (2005), `http://nusmv.irst.itc.it/`
14. Offutt, A.J., Pan, J.: Automatically detecting equivalent mutants and infeasible paths. Software Testing, Verification & Reliability 7(3), 165–192 (1997)
15. Purandare, M., Somenzi, F.: Vacuum cleaning CTL formulae. In: Proceedings of the 14th Conference on Computer Aided Design, pp. 485–499. Springer, Heidelberg (2002)
16. Rajan, A., Whalen, M.W., Heimdahl, M.P.E.: The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In: Proceedings of 30th International Conference on Software Engineering (ICSE) (to appear, 2008), `http://crisys.cs.umn.edu/ICSE08.pdf`
17. Rayadurgam, S.: Automatic Test-case Generation from Formal Models of Software. PhD thesis, University of Minnesota (November 2003)
18. Rayadurgam, S., Heimdahl, M.P.E.: Coverage based test-case generation using model checkers. In: Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001), April 2001, pp. 83–91. IEEE Computer Society Press, Los Alamitos (2001)
19. Rayadurgam, S., Heimdahl, M.P.E.: Generating MC/DC adequate test sequences through model checking. In: Proceedings of the 28th Annual IEEE/NASA Software Engineering Workshop – SEW 2003, Greenbelt, Maryland (December 2003)
20. RTCA. DO-178B: Software Consideration. In: Airborne Systems and Equipment Certification. RTCA (1992)
21. Whalen, M.W., Rajan, A., Heimdahl, M.P.E.: Coverage metrics for requirements-based testing. In: Proceedings of International Symposium on Software Testing and Analysis (July 2006)

Decomposition for Compositional Verification

Björn Metzler, Heike Wehrheim, and Daniel Wonisch

Universität Paderborn
Institut für Informatik
33098 Paderborn, Germany
{bmetzler,wehrheim,dwonisch}@uni-paderborn.de

Abstract. Compositional verification is based on the idea that the correctness check of a complex system can be divided into smaller verification tasks for its components. In this paper, we show how to *decompose* a specification into components when either no such decomposition is given, or when the given composition does not lend itself to an efficient compositional verification. Our decomposition is the starting point for an application of the L^* learning algorithm, generating assumptions for an assume-guarantee reasoning. We prove correctness of the decomposition as well as present experimental results using the model checker FDR2 as the teacher during learning.

1 Introduction

In formal system development verification ensures that the system meets the requirements set out by the designers or customers. Most often *model checking* is applied in the verification process to free the developer from manual proofs of correctness. Despite enormous progress made in this area ever since the invention of model checking [6], the problem of state explosion still hampers the verification of large systems. A lot of research today is still devoted to developing techniques which consequently allow model checking to scale to complex systems. Such methods range from symbolic model checking with BDDs or SAT techniques via symmetry or partial order reductions to various sorts of abstraction mechanisms.

One such technique – and the one we will be interested in here – is *compositional verification* [9]. Compositional verification takes a divide-and-conquer approach to checking correctness: instead of verifying the system as a whole, the system components are checked and the verification results are combined. One specific approach to compositional verification is *assume-guarantee (AG) reasoning* [13, 16, 19]. The verification of a system $S = S_1 \mid\mid S_2$ with respect to a property *Prop* is carried out in two steps: first, we show that S_2 guarantees *Prop* under an assumption A about its environment, and then S_1 is shown to guarantee this assumption. As a proof rule:

$$\frac{\langle A\rangle\ S_2\ \langle Prop\rangle \qquad \langle true\rangle\ S_1\ \langle A\rangle}{\langle true\rangle\ S_1 \mid\mid S_2\ \langle Prop\rangle}$$

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 105–125, 2008.

The drawback of this rule is the use of an assumption A which needs to be found before verification can proceed. Recently, a new technique for automatic generation of assumptions based on learning has been proposed [8]. This technique starts with a general assumption and uses a model checker as a teacher to progressively make this assumption more precise until it either matches the premises of the above proof rule or the property can be shown not to hold. The efficiency of the learning algorithm (and thus of the AG reasoning) depends on the actual decomposition of the system [7]; ideally the assumption A should be much smaller than the component S_1. Moreover, the technique relies on the *existence* of a structuring of the system into parallel components.

In this paper we will be concerned with *constructing* decompositions in case that (a) the system is not structured into parallel components, or (b) the existing structure does not lend itself to efficient assume-guarantee reasoning (e.g. because the assumption A gets too large). The starting point for our technique is a set of formal specifications written in CSP-OZ [11], a combination of the process algebra CSP [15] and the state-based formalism Object-Z [25]. The semantics of CSP-OZ are defined in terms of the semantic domain of CSP. Given a CSP-OZ specification, we first construct its *dependence graph* containing control flow as well as data dependencies among specification elements (here, Z schemas). The dependence graph construction follows a technique proposed in [5] for slicing CSP-OZ specifications. The graph is next *cut* into (currently two) parts. Roughly speaking, these two parts represent the two parallel components of the system; a definition of valid cuts and an appropriate synchronisation of the components has to ensure that the decomposition does not change the overall semantics of the specification. We consequently prove correctness of the decomposition. The cut determines the *interface* between components; by choosing a small cut we can produce small assumptions for AG reasoning.

The components we obtain through this decomposition are the starting point for the above sketched compositional verification, in which we use the technique proposed in [8] to *learn* the assumption. The employed L^* learning algorithm [1] for regular languages requires a teacher to answer membership and equivalence queries. In [8] the teacher is a model checker. As we are working in the semantic domain of CSP, we use the CSP model checker FDR2 as teacher and are thereby able to evaluate the effectiveness of the decomposition. It turns out that a compositional verification of our generated decomposition can outperform FDR2's performance during a non-compositional verification on the system as well as during a compositional verification starting with the given decomposition of the system. This is exemplified by a case study of a CSP-OZ specification of the Two-Phase-Commit Protocol and its natural – as well as generated – decomposition.

2 Background and Example

The running example for this paper on which we illustrate our decomposition as well as the verification is the Two-Phase-Commit Protocol (TPCP) [3]. We specify this protocol in CSP-OZ, an integrated formalism combining CSP and

Object-Z. While CSP is responsible for specifying the ordering of operations in the two phases of the protocol, Object-Z takes on the role of fixing what the operations themselves do.

The purpose of the protocol is to guarantee consistency of N local sites (or *pages*) of a distributed database. Instructed by a coordinator process, the protocol results in either all pages committing their transaction or all pages aborting it. As the name says the protocol works in two phases:

- *Phase 1: Commit-request*: The protocol starts with the coordinator process informing all participating pages about a *request* to commit the current transaction. Next, all pages *execute* the transaction and send a *vote* to the coordinator dependent on whether the local transaction succeeded (YES) or failed (NO). The coordinator collects the votes and *decide*s to either $COMMIT$ in the case that all votes agree on YES, or to $ABORT$ the transaction.
- *Phase 2: Commit*: The coordinator *inform*s all pages about the decision. All participating sites behave accordingly: an abort leads to an *undo* of the transaction while a commit leads to *complet*ion. In any case, the sites output the *result* and send an *acknowledge*ment to the coordinator.

Let N be the number of pages participating in the protocol and let *Votes* and *Trans* be the following two base types:

$Votes == \{YES, NO\}$
$Trans == \{COMMIT, ABORT\}$

The specification given below is the CSP-OZ class for the central coordinator.

Coord

method $request, vote[x? : Votes], decide, inform[x! : Trans], acknowledge$

main $\stackrel{c}{=} |||_{0<i\leq N}(request \rightarrow \texttt{Skip});\ |||_{0<i\leq N}(vote \rightarrow \texttt{Skip});$
$decide \rightarrow |||_{0<i\leq N}(inform \rightarrow \texttt{Skip});\ |||_{0<i\leq N}(acknowledge \rightarrow \texttt{Skip})$

$dec : Trans$
$votes : \mathbb{P}\ Votes$

Init

$dec = ABORT$
$votes = \varnothing$

request

$\Delta(votes)$

$votes' = \varnothing$

acknowledge

vote

$\Delta(votes);\ vo? : Votes$

$votes' = votes \cup \{vo?\}$

inform

$in! : Trans$

$in! = dec$

decide

$\Delta(dec)$

if $(NO \in votes)$ **then** $dec' = ABORT$ **else** $dec' = COMMIT$

The first part of the class defines its interface, i.e. the methods/operations (or channels in the CSP terminology) with their signatures. The next part is a CSP process equation describing the ordering of operations of the coordinator. Class *Coord* starts by sending a request to all N pages, accepts all the votes, decides, and consequently informs all pages about the decision, and finally waits for an acknowledge. This ordering is specified using the CSP operators for interleaving ($|||$) – a special form of parallel composition – sequencing ($;$), prefixing of operations ($\rightarrow$) and the empty terminating process *Skip*. The third part – the Object-Z part – consists of a number of schemas specifying the class' state space, initialisation and the operations. The class has two variables: *dec* for holding the final decision and a set of votes *votes*. The operations can or cannot modify these variables (specified by the Δ-list) and input and output variables (marked ? and !, respectively) help to pass values between classes. For instance, operation *vote* stores an input *vo*? in the variable *votes*. Input and output variables are in general not restricted by the CSP part. Therefore, we refrain from denoting them there.

Note that an empty schema describes an operation which leaves all variable values unchanged. In the following we will leave out empty schemas. Both parts, CSP as well as Object-Z, impose restrictions on the behaviour of the class which need to be jointly obeyed.

The class *Coord* operates in parallel with several instantiations of the following class *Page*.

Page

`method` *request*, *execute*, *vote*$[x! : Votes]$, *inform*$[x? : Trans]$
`method` *undo*, *complete*, *result*$[b! : Trans]$, *acknowledge*

`main` $\stackrel{c}{=}$ *request* $\rightarrow$ *execute* $\rightarrow$ *vote* $\rightarrow$ *inform* $\rightarrow P$
$P \stackrel{c}{=}$ *undo* $\rightarrow$ *result* $\rightarrow$ *acknowledge* $\rightarrow$ `Skip`
$\quad \Box$ *complete* $\rightarrow$ *result* $\rightarrow$ *acknowledge* $\rightarrow$ `Skip`

$dec : Trans$
$stable : \mathbb{B}$

Init

$dec = ABORT$
$stable$

result

$b! : Trans$

$b! = dec$

execute

$\Delta(stable)$

$stable' \in \{\text{true}, \text{false}\}$

undo

$dec = ABORT$

complete

$dec = COMMIT$

inform

$\Delta(dec)$
$in? : Trans$

$dec' = in?$

vote

$vo! : Votes$

$stable \Rightarrow vo! = YES$
$\neg stable \Rightarrow vo! = NO$

Here, we employ an additional CSP operator: the choice operator ($\Box$) for choosing between alternatives. The choice between *undo* and *complete* is determined by the Object-Z part: *undo* has a predicate $dec = ABORT$ in its schema (which works as a precondition), while *commit* can only be executed when *dec* equals $COMMIT$.

The full system is specified as

$$System = Coord \|_S (|||_{0<i\leq N} Page)$$

with $S = \{request, vote, inform, acknowledge\}$. Here, we use a different CSP parallel composition operator: $\|_S$ requires joint execution of the operations (or events in CSP terminology) in S, i.e. *Coord* and *Pages* need to synchronize on *request*, *vote*, *inform* and *acknowledge*. For the remainder of this paper, let $Pages = |||_{0<i\leq N} Page$. This completes the specification of the TPCP.

Next, we are interested in verifying a specific property of the Two-Phase-Commit protocol, i.e. of *System*. The property is a safety property and states that if at least one page votes *NO*, all pages will *undo* the transaction. Before formally specifying this property, we first have to clarify the language we use for writing properties. The formalism CSP-OZ has a joint semantics for CSP and Object-Z parts, which is given in terms of CSP alone. CSP on the other hand has a semantics given within the *failures-divergences* model. Here, we will solely be interested in safety properties and move to the simpler domain of *traces*. Traces represent the behaviour of a system in terms of the possible orderings of its events. Thus the basis for our semantics is a set *Events* representing all valid events of our system. This set consists of operation names together with values for parameters. For our example, *Events* contains the events $inform.ABORT$, *decide*, $vote.NO$, etc. Given this set *Events*, the trace semantics of a system S is a (prefix-closed) set of traces:

$$traces(S) \subseteq 2^{Events^*}$$

As with the systems, we also specify properties as sets of traces, namely simply by giving all valid traces possessing a particular property. While we could also use CSP-OZ for property specification, here we will stick to CSP for this purpose. The following CSP specification presents our correctness property for the TPCP:

$$\begin{aligned}
PROP &= PC(N) \\
PC(0) &= |||_{0<i\leq N} complete \rightarrow \texttt{Skip} \\
PC(j) &= vote.YES \rightarrow PC(j-1) \,\Box\, vote.NO \rightarrow PU(j-1) \\
PU(0) &= |||_{0<i\leq N} undo \rightarrow \texttt{Skip} \\
PU(j) &= \Box_{x:\{YES,NO\}} vote.x \rightarrow PU(j-1)
\end{aligned}$$

The process $PC(j)$ allows for j votes (and thus *PROP* for N votes) and - if control flow has not left the process before - finally N events *complete*. As soon as one vote is *NO*, PC processes switch to some process *PU*. $PU(j)$ also allows for j votes (with any parameter value) but always terminates with *undo*s. Thus

as soon as we have one event *vote.NO*, the final events of *PROP* will be *undo*. The question is now whether the traces of *System* (concerning the interesting events *vote*, *undo* and *complete*, here extracted by hiding (\) all other events) are contained in those of *PROP*, i.e.

$$traces(System \backslash \{request, execute, inform, decide, result, acknowledge\}) \subseteq traces(PROP)$$

This check for trace inclusion is a standard check for CSP specifications as one of CSP's refinement orderings is trace inclusion.

Definition 1. *Let P, Q be CSP processes. P is a* trace refinement *of Q, if $traces(P) \subseteq traces(Q)$. We write $Q \sqsubseteq_T P$. P is* trace equivalent *to Q, $P =_T Q$, if $P \sqsubseteq_T Q$ and $Q \sqsubseteq_T P$.*

This being standard for CSP, we can use the CSP model checker FDR2 [18] to check it, using a technique proposed in [12] to translate a CSP-OZ specification into CSP. As it turns out, FDR2 fails to carry out this check for more than 5 pages.

Next, we tried to use assume-guarantee reasoning to show the property. The *System* specification is already structured using parallelism, with *Coord* being S_1 and all *Pages* S_2. Thus we used the AG rule given in the introduction. This rule has been proven correct and complete [21]. Rephrased in terms of CSP's traces refinement, the rule reads:

$$\frac{PROP \sqsubseteq_T A \,||_X\, S_2 \qquad A \sqsubseteq_T S_1}{PROP \sqsubseteq_T S_1 \,||_Y\, S_2} \tag{1}$$

Here, the sets X and Y are synchronisation sets defined by $X = \alpha(A) \cap \alpha(S_2)$ and $Y = \alpha(S_1) \cap \alpha(S_2)$ (intersection of alphabets). The assumption A represents a restriction on S_2's environment which is necessary for S_2 to guarantee *PROP*. On the other hand, S_1 needs to guarantee this restriction. For generating the assumption we use the technique proposed in [8]. This method employs Angluin's L^* algorithm for learning regular languages (a finite automaton) to generate the assumption A. The algorithm needs to employ a teacher which can answer membership as well as equivalence queries. The proposal of [8] was to use a model checker to this end. We have a prototypical implementation of the L^* algorithm which takes CSP processes as inputs, calls the CSP model checker FDR2 whenever a teacher is required and ultimately either outputs 'true' and an assumption if $PROP \sqsubseteq_T S_1 \,||_Y\, S_2$ is true or 'false', if not. Using the given structure of *System*, the assume-guarantee reasoning unfortunately gives us no gain at all. In the contrary, the run-times get worse and the check fails as soon as we reach 5 pages (see the section on experimental results). The reason for this is that the part which produces most of the complexity is *Pages* as it is the interleaving of a large number of processes. The state space of *Pages* needs to be constructed in both the verification of the complete system and in the

AG verification. It would be preferable to have a decomposition of the system which *splits* the process *Pages* such that the compositional verification never needs to consider *Pages* in its entirety. Next, we will see how to construct such a decomposition of the system.

3 Decomposition of a Specification

We aim to decompose a specification into two components allowing the application of assume-guarantee reasoning. To find a suitable decomposition in this context we need to analyse a specification's *dependence structure*: the specification's elements (operations of a class) might depend on each other. The distribution of dependent elements over both components is not desirable. However, if required, it necessitates that an assumption describes the correlation between the different operations. Thus we need to define what *dependence* means and we need to ensure that the decomposition preserves the overall dependence structure of the specification. To find a small assumption – and this is preferable – the number of intersecting dependencies between the components should be small.

Fortunately, for CSP-OZ and in the context of program slicing [28], Brueckner [4] developed a precise dependence analysis for CSP-OZ and defined a specification's *Dependence Graph*:

Definition 2. *(Dependence Graph of a specification)*
The Dependence Graph *(DG)* $G = (N, \rightarrow_{DG})$ *of a specification* S *is defined over a set of nodes* $N = cf(N) \cup op(N)$ *and a set of edges* $\rightarrow_{DG} = \rightarrow \cup \dashrightarrow$. *The set* $cf(N)$ *corresponds to operators within the specification's CSP part whereas* $op(N)$ *corresponds to operations of the specification.*[1] *For the set of edges, we distinguish control flow edges* ($\rightarrow$) *from program dependence edges* ($\dashrightarrow$).

The set of DG edges describes dependencies between different nodes of the DG mostly following the principle of cause and effect – i.e. the edge's source node controls or influences execution of its target node. The DG comprises the *Control Flow Graph (CFG)* and the *Program Dependence Graph (PDG)*.

As the name says, the CFG covers dependencies with respect to the specification's control flow structure. This is mainly derived from its CSP part. As an example: in class *Page*, *request* prefixes *execute* leading to a control flow edge from *request* to *execute*.

Along with this, the PDG edges describe dependencies between different operations of the specification such as data-, control-, synchronisation data- or interference data dependencies. They refer to the set of state variables of the class' Object-Z part. An example of a data dependence is the edge from *execute* to *vote* – the variable *stable* is modified within *execute* and referenced in *vote*, i.e. $stable \in mod(execute) \cap ref(vote)$ where $mod(op)$ and $ref(op)$ denote the

[1] For simplicity, instead of defining the DG with respect to predicate nodes, we use operation nodes.

sets of *modified* and *referenced* variables within an operation *op*, respectively. A synchronisation data dependence exists between the events *execute* of both classes since *Page.execute* has an output that *Coord.execute* uses as an input. Note that PDG edges only connect operation nodes.

As a multiple occurrence of an Object-Z operation within the CSP part of a specification is possible, we define the correlation between operation nodes of the DG and operations of the specification:

Definition 3. *(Labelling of DG nodes)*
Let $G = (N, \rightarrow_{DG})$ *be the DG of a specification* S *and let* Op *be the set of all operations of* S*. The* labelling function $l : op(N) \rightarrow Op$ *maps an operation node of the DG on its corresponding operation name within* S*. For* $O \subseteq Op$*, we define* $l^{-1}[O] := \{n \in op(N) \mid l(op) \in O\}$.

In the following, we assume the alphabet of the CSP part and the set of Object-Z operations to be equal. Thus, for the CSP part, *Op* is the set of events projected on its *names* omitted from its parameters.

For a complete definition of a specification's DG, see [4]. Figure 1 shows (a slightly simplified version of[2]) the DG for *System*. We use different types of arrows to illustrate control flow- (→) and program dependence-edges (⇢).

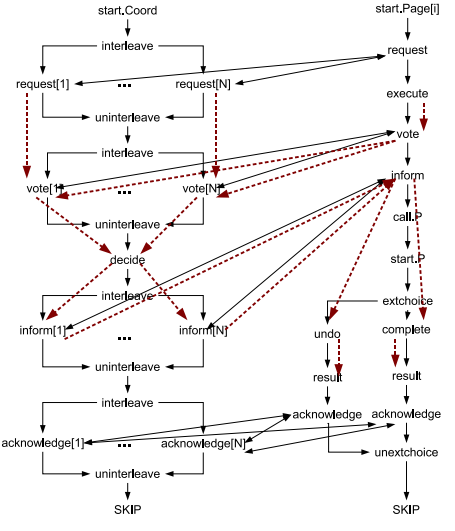


Fig. 1. Dependence Graph for *System*

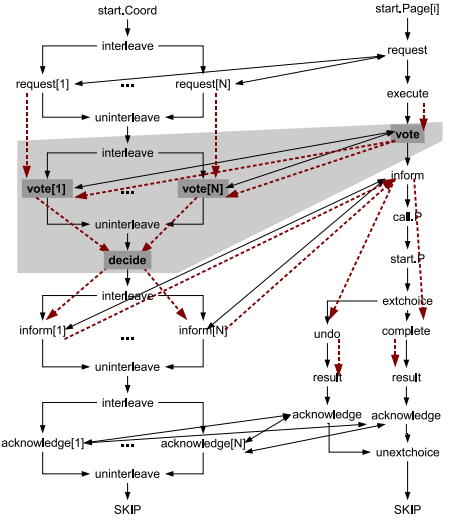


Fig. 2. Cut for $l^{-1}[\{decide, vote\}]$

[2] Here, we omit the additional *start*- and *term*-nodes for the parallel composition of both classes. We do also not incorporate the intermediate *seq*-nodes within *Coord* for the sequential composition of the interleavings. For an operation *op*, we use *op*[*i*] to depict its *i*-th execution within the corresponding CSP-interleaving. *Page*[*i*] is an arbitrary instance of *Page*.

3.1 A Cut of a Dependence Graph

All dependencies between different specification elements are represented in the specification's DG. Thus to decompose a given specification S into two parts, we start by defining a decomposition of the DG.

The basic idea is the definition of a *cut* $\mathbf{C}$ identifying the interface between the parallel components which we define subsequently. Being a subset of the DG's operation nodes, a cut fragments the DG into two subgraphs representing the two stages (phases) of the graph. The cut will then yield a decomposition of the specification itself. In the context of assume-guarantee reasoning, we set the following objectives:

- The overall semantics of S are preserved, i.e. the original specification is trace equivalent to its decomposition, when both parts are combined via parallel composition,
- the decomposition is efficient in the context of assume-guarantee reasoning, i.e. a cut leads to a relatively small intersection between the components and uniformly distributed operations.

In the remainder of the paper, we will not generally distinguish between control flow edges and program dependence edges. Next, we define the cut of a specification's dependence graph.

Definition 4
Let $G = (N, \rightarrow)$ be a graph and $N' \subseteq N$. Then,

$$N'\downarrow := \{n \in N \mid \exists\, n' \in N' \bullet n \rightarrow^* n'\} \text{ (all nodes reaching } N'\text{)},$$
$$N'\uparrow := \{n \in N \mid \exists\, n' \in N' \bullet n' \rightarrow^* n\} \text{ (all nodes reachable from } N'\text{)}.$$

Definition 5. *(Cut of the DG)*
Let $G = (N, \rightarrow_{DG})$ be the DG of a given specification. A cut $\mathbf{C} \subseteq op(N)$ *of G is a subset of the operation nodes of N such that*

a) $\mathbf{C}\downarrow \cup \mathbf{C}\uparrow = N$,
b) $\nexists c \in \mathbf{C}\downarrow \setminus \mathbf{C}, c' \in \mathbf{C}\uparrow \setminus \mathbf{C} \bullet c \rightarrow_{DG} c' \vee c' \rightarrow_{DG} c$,
c) $\forall\, n_1, n_2 \in op(N) \bullet l(n_1) = l(n_2) \Rightarrow (n_1 \in \mathbf{C} \Leftrightarrow n_2 \in \mathbf{C})$

$\mathbf{C}$ is a subset of the DG's operation nodes determining a split of the DG into $\mathbf{C}\downarrow$ and $\mathbf{C}\uparrow$ and it is used to define the decomposition of a specification S into S_1 and S_2. S_1 and S_2 are the parallel components of the decomposition with the intersection of $\mathbf{C}\downarrow$ and $\mathbf{C}\uparrow$ defining the interface between S_1 and S_2.

Condition a) states that for any node $n \in N$, either the cut is reachable from n or n is reachable from the cut. Therefore, no node will be left out. *Condition b)* states that DG edges must not cross the cut. This condition ensures that there are no dependencies from one component to another circumventing the cut. *Condition c)* states that for operations with multiple occurrence within the CSP part, either *all* or *none* of the corresponding DG nodes are contained in the cut. This condition is required to ensure proper synchronisation between the constructed parallel components.

In the context of a cut $\mathbf{C}$, we will call $\mathbf{C}\downarrow$ the *precut* and $\mathbf{C}\uparrow$ the *postcut*. Next, we define a condition on the relation between $\mathbf{C}\downarrow$ and $\mathbf{C}\uparrow$ to restrict the distribution of a DG:

Definition 6. *(sequential cut)*
Let $G = (N, \rightarrow_{DG})$ be the DG of a given specification. A cut $\mathbf{C}$ ***sequentially distributes*** *G, iff $\mathbf{C}\uparrow \cap \mathbf{C}\downarrow = \mathbf{C}$. We call $\mathbf{C}$ a* sequential cut*.*

The condition for the sequential distribution of a DG states that there are no nodes leading to the cut which are also reachable from the cut. Thus, the DG can be viewed in two stages, with a unique distribution of all nodes: a first stage before the cut and a second stage starting from the cut, with the cut itself being their intersection. In particular, a sequential cut requires that all cycles of the DG are distributed over the resulting two subgraphs without intersecting with the cut itself. All paths connecting both subgraphs must pass the cut.

Even though we consider a rather specific, simple class of dependence graphs to illustrate our approach and the applicability of the assume-guarantee proof rule, our approach is not restricted to sequential distributions: the definition of a cut can as well be applied to circular dependence graphs. This will be part of our future work.

For our example, based on two heuristics for the definition of a cut, the decomposition of the DG with respect to the set $\mathbf{C} = l^{-1}[\{vote, decide\}]$[3] is given in Figure 2. These heuristics can informally be described as follows:[4]

- A cut should contain as few as possible nodes and its corresponding operations should modify as few as possible variables,
- A cut should be defined in the middle part of the DG.

Using the set $\mathbf{C}' = \{decide\}$ would lead to a violation of the cut definition due to the cut-crossing CFG edge from *vote* to *inform* on the right hand side of the DG. Therefore, we additionally needed to add *vote*. $\{decide, vote\}$ indeed defines a sequential cut since neither there are cut-crossing edges nor nodes outside of $\mathbf{C}$ assigned to the first and to the second stage. Condition c) holds as well since all DG nodes assigned to the operation *vote* are contained in the cut.

3.2 Decomposition of a Specification

As a next step, we define the decomposition of a specification. This will be done with respect to a sequential cut of its DG. Precut and postcut will be used to define two components S_1 and S_2 with the following goal: $S_1 \mid\mid S_2$ has the same set of traces as S and is therefore – in our semantic domain – equivalent to S. The decisive point in this definition is the synchronisation alphabet: we need to guarantee correct values for the state variables in the second stage. These variables might have been modified during the first stage. Synchronization should

[3] In the following cut-examples, we will synonymously use S and $l^{-1}[S]$.
[4] A closer investigation of heuristics for selecting optimal cuts will form part of our future work.

thus lead to a passing of the current values. To ensure this, we use the set of cut events as the synchronisation alphabet and identify all variables modified during the first stage. These are then communicated to the second stage.

A CSP-OZ definition of a class S consists of the following elements:

S	
I	[interface definition]
`main`	[CSP part]
State	[OZ part: state schema]
Init	[OZ part: initial state schema]
op	[OZ part: operations]

For $m \in Op$, a method declaration has the form $m[p_1 : t_1, \ldots, p_m : t_m]$ with parameters p_i of type t_i. For the corresponding Object-Z operation, *op.par* denotes its parameter declaration and *op.pred* its predicate part. *Var* denotes the set of state variables of a class. For $M \subseteq Op$, let $\{|\ M\ |\} := \{m.i.o \in Events \mid m \in M\}$.

To define the decomposition, we first need to define a projection of a CSP process to a subset of its events. This projection will then be used to decompose the CSP part with respect to the precut and the postcut.

Definition 7. *(Projection of CSP processes, [4])*
Let P be the right-hand side of a CSP process definition and $E \subseteq Events$. The projection of P on E, *denoted by $P|_E$, is inductively defined:*

1. $\texttt{Skip}|_E := \texttt{Skip}$ *and* $\texttt{Stop}|_E := \texttt{Stop}$,
2. $(e \to P)|_E := \begin{cases} P|_E, & e \notin E \\ e \to P|_E, & \textit{otherwise}, \end{cases}$
3. $(P \circ Q)|_E := (P|_E) \circ (Q|_E)$ *for* $\circ \in \{;\ , |||, \Box, \sqcap\}$,
4. $(P \mid\mid_S Q)|_E := (P|_E) \mid\mid_{S \cap E} (Q|_E)$.

To determine the projection of a complete CSP part, Definition 7 has to be applied to every CSP process definition. Next, we define the decomposition of S with respect to a sequential cut:

Definition 8. *(Decomposition with respect to a sequential Cut)*
Let S be a specification and $G = (N, \to_{DG})$ be its dependence graph. Let $\mathbf{C}$ be a sequential cut and let $M_1 := l[\mathbf{C}\downarrow \cap op(N)]$, $M_2 := l[\mathbf{C}\uparrow \cap op(N)]$, $M_\mathbf{C} := M_1 \cap M_2$, $E_i := \{|\ M_i\ |\}$, $E_\mathbf{C} := \{|\ M_\mathbf{C}\ |\}$, $V_1 := Var(M_1)$, $V_2 := Var(M_2)$ and

$$V_\mathbf{C} = \{x \in S.Var \mid \exists\, n \in M_\mathbf{C}, n' \in (M_2 \setminus M_\mathbf{C}) \bullet n \to^*_{DG} n' \wedge x \in (mod(n) \cap ref(n'))\}.$$

Given a set $V_\mathbf{C} = \{x_1, \ldots, x_n\}$ of types s_i, for $m \in Op$, let $\{x_{m_1}, \ldots, x_{m_k}\} = V_\mathbf{C} \cap mod(m)$. We use a function f to define the interface extension of the class:

$$f(m[p_1 : t_1, \ldots, p_m : t_m] = \begin{cases} m[p_1 : t_1, \ldots, p_m : t_m, a_{m_1} : s_{m_1}, \ldots, a_{m_k} : s_{m_k}, b_{m_1} : r_{m_1}, \ldots, b_{m_l} : r_{m_l}], & m \in M_\mathbf{C} \\ m[p_1 : t_1, \ldots, p_m : t_m], & \textit{otherwise} \end{cases}$$

The decomposition of S with respect to $\mathbf{C}$ into S_1 and S_2 *is defined as*[5]*:*

S_1

$I_1 := f(I|_{M_1})$

$\texttt{main}_1 := \texttt{main}|_{E_1}$ /extended by additional parameters/

$\texttt{State}_1 := \texttt{State} \upharpoonright V_1$

$\texttt{Init}_1 := \texttt{Init} \upharpoonright V_1\ (*)$

$op_1 :=$

$$\begin{cases} op, & op \in M_1 \setminus M_{\mathbf{C}} \\ [op.par, a_{m_i}! : s_{m_i}, b_{m_j}? : r_{m_j} \mid op.pred \wedge \bigwedge_{i=1}^{k} a_{m_i}! = x'_{m_i}], & op \in M_{\mathbf{C}} \end{cases}$$

S_2

$I_2 := f(I|_{M_2})$

$\texttt{main}_2 := \texttt{main}|_{E_2}$ /extended by additional parameters/

$\texttt{State}_2 := \texttt{State} \upharpoonright V_2$

$\texttt{Init}_2 := \texttt{Init} \upharpoonright V_2\ (*)$

$op_2 :=$

$$\begin{cases} op, & op \in M_2 \setminus M_{\mathbf{C}} \\ [op.par, a_{m_i}? : s_{m_i}, b_{m_j}? : r_{m_j} \mid \bigwedge_{i=1}^{k} x'_{m_i} = a_{m_i}?], & op \in M_{\mathbf{C}} \end{cases}$$

$I|_M$ depicts the restriction of the set of methods within the interface I onto M. For the initial state schemas $\texttt{Init}_i$, the definition is annotated with an asterisk: the coarse idea is a projection of `Init` onto all predicates solely dealing with V_i. Atomic predicates sharing variables local to S_1 and S_2 need to be restricted but can not be left out. Here, we omit a detailed definition of $\texttt{Init}_i$.

We take a closer look at the additional parameters for $m \in M_{\mathbf{C}}$. Firstly, $V_{\mathbf{C}}$ defines exactly the set of state variables from the first stage that influence the second stage of the specification. Note that any such variable must be modified *inside* some cut event since otherwise there would be cut-crossing edges in the DG. For each $x_i \in V_{\mathbf{C}} \cap mod(op)$, we use parameters $a_{m_i} : s_{m_i}$ extending the type of op. These parameters uncover the influence of the first component on the second one. Secondly, Definition 5 allows for a cut containing two or more DG nodes with the same labelling. Since parallel composition based on events does not distinguish between these nodes, we need to ensure that in the decomposition, corresponding instances of the event are synchronized. This is achieved by adding additional *address parameters* $b_{m_j} : r_{m_j}$ to the respective channels solely being restricted by the CSP part. On the one hand, these parameters ensure that all previously allowed synchronisations are still possible. On the other hand, synchronisation between S_1 and S_2 is restricted to matching DG nodes. The number of address parameters depends on the number of classes synchronizing on op whereas the type r_{m_j} depends on the cardinal number of $l^{-1}[\{op\}]$. We will exemplify this on our example and refrain from giving a precise definition here.

For the Object-Z part, we extend any operation of the cut with corresponding additional outputs (S_1) and inputs (S_2), respectively. Moreover, we eliminate the remaining predicate part of the shared operations within S_2.

[5] $\texttt{State} \upharpoonright V$ denotes the projection of *State* on a subset V of its state variables.

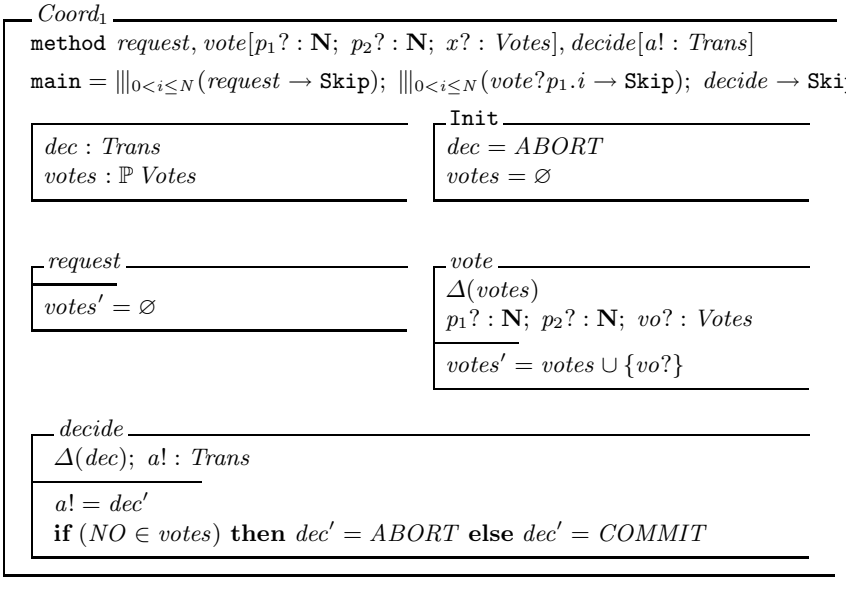
3.3 Example Revisited

We apply the decomposition to our example. Based on our objective to decompose a specification into equally distributed components, we define several sequential cuts and investigate each one. The verification results are given in the next section. Here, we illustrate the decomposition for the set $E_{\mathbf{C}} = \{vote, decide\}$ as introduced in Section 3.1. We split E into

$E_1 = \{request, execute, vote, decide\}$ and
$E_2 = \{vote, decide, inform, undo, complete, failure, result, acknowledge\}$.

The definition of a DG's cut is not restricted to a single class. In fact, the DG is defined with respect to the full specification possibly containing several classes. Since *System* comprises two classes *Coord* and *Page*, both will be decomposed into components $Coord_1, Coord_2$ and $Page_1, Page_2$, respectively. For the parallel composition $System_1 \,||_{E_{\mathbf{C}}}\, System_2$, $System_i$ is defined as $Coord_i \,||\, Pages_i$. $System_1$ can be viewed as the first and $System_2$ as the second stage of the protocol. Control flow according to *System* is restored by the parallel composition.

According to Definition 8, the resulting specification slices are given below. Since $V_{\mathbf{C}} = \{Coord.dec\}$, event *decide* is extended by one parameter for the communication of *Coord.dec*. Also, event *vote* is extended by two additional parameters to ensure synchronisation for matching occurrences of *vote*. We explicitly depict these parameters in the CSP part since they are restricted there. Based on $|\, l^{-1}[\{Coord.vote\}] \,|=|\, l^{-1}[\{Pages.vote\}] \,|= N$, these parameters are of type $\mathbf{N} = \{1, \ldots, N\}$. To address specific instances of $Page_1$ and $Page_2$, we adopt CSP-OZ's concept of constant parameters, and use $Pages_j = |||_{0<i\leq N} Page_j(i)$.



Page$_1(i : \mathbf{N})$

`method` *request*, *vote*$[p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ x! : Votes]$, *execute*

`main` = *request* $\rightarrow$ *execute* $\rightarrow$ *vote*.$i?p_2 \rightarrow$ `Skip`

stable : $\mathbb{B}$

`Init`

stable

vote

$p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ vo! : Votes$
$stable \Rightarrow vo! = YES$ $\neg stable \Rightarrow vo! = NO$

execute

$\Delta(stable)$
$stable' \in \{\text{true}, \text{false}\}$

Coord$_2$

`method` *vote*$[p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ x? : Votes]$, *decide*$[a? : Trans]$

`method` *inform*$[x! : Trans]$, *acknowledge*

`main` = $|||_{0<i\leq N}(vote?p_1.i \rightarrow$ `Skip`$);$ *decide* $\rightarrow |||_{0<i\leq N}(inform \rightarrow$ `Skip`$);$
$|||_{0<i\leq N}(acknowledge \rightarrow$ `Skip`$)$

dec : *Trans*

`Init`

$dec = ABORT$

inform

$in! : Trans$
$in! = dec$

decide

$\Delta(dec);\ a? : Trans$
$dec' = a?$

vote

$p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ vo? : Votes$

Page$_2(i : \mathbf{N})$

`method` *vote*$[p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ x! : Votes]$, *inform*$[x? : Trans]$, *undo*

`method` *complete*, *result*$[b! : Trans]$, *acknowledge*

`main` = *vote*.$i?p_2 \rightarrow$ *inform* $\rightarrow P$

P = *undo* $\rightarrow$ *result* $\rightarrow$ *acknowledge* $\rightarrow$ `Skip`
$\Box$ *complete* $\rightarrow$ *result* $\rightarrow$ *acknowledge* $\rightarrow$ `Skip`

dec : *Trans*

`Init`

$dec = ABORT$

inform

$\Delta(dec);\ in? : Trans$
$dec' = in?$

result

$b! : Trans$
$b! = dec$

undo

$dec = ABORT$

complete

$dec = COMMIT$

vote

$p_1? : \mathbf{N};\ p_2? : \mathbf{N};\ vo! : Votes$

In [9], the motivation for introducing and specifying the TPCP is its particular structure allowing for an appliance of the *Communication-Closed-Layers law* (CCL) [10]. Our way of decomposing a specification is one particular way of adopting the CCL.

3.4 Correctness of the Decomposition

We will now show that the full specification is trace equivalent to the composition of both components constructed in Definition 8. As mentioned in Section 2, for our main goal we want to apply the assume-guarantee rule 1 from Section 2 to show $PROP \sqsubseteq_T S$. To show correctness of the decomposition we have to show

$$S =_T S_1 \,||_{E_{\mathbf{C}}}\, S_2, \tag{2}$$

i.e. our original specification is trace equivalent to the parallel composition of the components. Then we can apply the given rule with respect to S:

$$\frac{PROP \sqsubseteq_T A \,||_X\, S_2 \qquad A \sqsubseteq_T S_1}{PROP \sqsubseteq_T S}$$

The following lemma will be applied to establish the overall correlation between the specification and the decomposition:

Lemma 1
Let P_i, Q_i be CSP processes and A_i, B_i their respective alphabets. Then,

$$(P_1 \;{}_{A_1}||_{A_2}\; P_2) \;{}_{A_1 \cup A_2}||_{B_1 \cup B_2}\; (Q_1 \;{}_{B_1}||_{B_2}\; Q_2) =$$
$$(P_1 \;{}_{A_1}||_{B_1}\; Q_1) \;{}_{A_1 \cup B_1}||_{A_2 \cup B_2}\; (P_2 \;{}_{A_2}||_{B_2}\; Q_2)$$

Proof
We use rule (2.5)

$$(P \,||_{X \cap Y}\, Q) \,||_{(X \cup Y) \cap Z}\, R = P \,||_{X \cap (Y \cup Z)}\, (Q \,||_{Y \cap Z}\, R)$$

from [23], p. 57 and incrementally deduce the equation. □

Next, we state the main theorem of this paper: the decomposition of a specification based on a sequential cut is trace equivalent to the original specification.

Theorem 1. *(Correctness of the Decomposition)*
Let S be a specification and $G = (N, \rightarrow_{DG})$ be its DG. Let $\mathbf{C}$ be a sequential cut and let S_1 and S_2 be the decomposition of S with respect to Definition 8. Then, the following holds:

$$S =_T S_1 \,||_{E_{\mathbf{C}}}\, S_2 \tag{3}$$

Proof
In our semantic domain we are interested in $traces(S) \subseteq 2^{Events^*}$. Based on the CSP trace semantics for CSP-OZ we get $traces(S) := traces(\texttt{main} \,||_{Events}\, OZ)$. Thus, a trace of a CSP-OZ class is a trace within the parallel composition of the specification's CSP part and Object-Z part, respectively, synchronizing on the set *Events*. We compositionally show (3) by dealing with the specification's CSP- and Object-Z part independently. If we can show

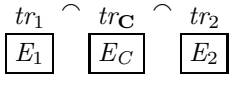
$$\texttt{main} =_T \texttt{main}_1 \;||_{E_\mathbf{C}}\; \texttt{main}_2, \tag{4}$$

$$OZ =_T OZ_1 \;||_{E_\mathbf{C}}\; OZ_2 \text{ for the set of traces of the CSP part,} \tag{5}$$

we can subsequently apply Lemma 1 with respect to $A_1 = A_2 = E_1$, $B_1 = B_2 = E_2$ and deduce

$$\begin{aligned}
& traces(S) \\
&= traces(\texttt{main} \;||_{Events}\; OZ) && (Def.\ of\ S) \\
&= traces((\texttt{main}_1 \;||_{E_\mathbf{C}}\; \texttt{main}_2) \;||_{Events}\; (OZ_1 \;||_{E_\mathbf{C}}\; OZ_2)) && ((4),(5)) \\
&= traces((\texttt{main}_1 \;||_{E_1 \cap E_2}\; \texttt{main}_2) \;||_{Events}\; (OZ_1 \;||_{E_1 \cap E_2}\; OZ_2)) && (E_1 \cap E_2 = E_\mathbf{C}) \\
&= traces((\texttt{main}_1 \;||_{E_1}\; OZ_1) \;||_{E_1 \cap E_2}\; (\texttt{main}_2 \;||_{E_2}\; OZ_2)) && (Lemma\ 1) \\
&= traces((\texttt{main}_1 \;||_{E_1}\; OZ_1) \;||_{E_\mathbf{C}}\; (\texttt{main}_2 \;||_{E_2}\; OZ_2)) && (E_1 \cap E_2 = E_\mathbf{C}) \\
&= traces(S_1 \;||_{E_\mathbf{C}}\; S_2) && (Def.\ of\ S_1, S_2)
\end{aligned}$$

Due to lack of space, we refrain from giving the complete proofs of (4) and (5) but outline the ideas. The core idea for (4) is to assume that any trace $tr \in traces(\texttt{main})$ has the following structure:



We then show that $tr \in traces(\texttt{main})$ if and only if $tr_1 \frown tr_\mathbf{C} \in traces(\texttt{main}_1)$ and $tr_\mathbf{C} \frown tr_2 \in traces(\texttt{main}_2)$ holds. Here, we particularly use Condition c) of Definition 5. However, if tr switches between different interleaving branches of the CSP part, an event of $E_2 \setminus E_\mathbf{C}$ can be executed before an event of $E_1 \setminus E_\mathbf{C}$ without violating the cut definition. To solve this problem, we apply Lemma 1 to restructure the trace and treat interleaving branches separately.

For the Object-Z part, $OZ =_T OZ_1 \;||_{E_\mathbf{C}}\; OZ_2$ would be preferable. However, this equivalence does in general not hold: if the CSP part does not determine the ordering of events, a trace within $traces(OZ)$ may not correspond to the paths of the DG. The cut is defined with respect to the DG, the decomposition might access and output inconsistent values. Thus, $tr \in traces(OZ)$ does not need to be an element of $traces(OZ_1 \;||_{E_\mathbf{C}}\; OZ_2)$ and vice versa. Indeed, we show the following, weaker property for the Object-Z part:

$$\forall\, tr \restriction Op \in traces(\texttt{main}) \restriction Op \bullet tr \in traces(OZ) \Leftrightarrow tr \in traces(OZ_1 \;||_{E_\mathbf{C}}\; OZ_2) \tag{6}$$

It states that $OZ =_T OZ_1 \;||_{E_\mathbf{C}}\; OZ_2$ if the CSP part determines the ordering of events within the trace. Since this will always be the case for a trace within a CSP-OZ specification, it is sufficient to show (5). To do so, given $tr \in traces(OZ)$, we need to construct $tr_i\ traces(OZ_i)$ such that both synchronize on $E_\mathbf{C}$. For the reverse direction, we have to construct an equivalent $tr \in traces(OZ)$ out of $tr_i \in traces(OZ_i)$.

The complete proof of (4) and (5) uses all the various dependencies of the PDG. For instance, data dependencies ensure that in case a certain variable is modified, it always refers to the correct variable values used in the modification. Synchronization dependencies ensure that synchronized events are not split between the two components. □

4 Implementation and Experimental Results

To evaluate our approach we implemented Angluin's learning algorithm and the framework of [8] for the CSP model checker FDR2 (Failure Divergence Refinement) [18] to automatically verify a specification against a property based on the assume-guarantee proof rule [29]. Using the CSP semantics of CSP-OZ developed in [12], the specification is translated into the input language of FDR2.

The property we are aiming at can be described as follows: if at least one page votes with *NO*, all pages will *undo* their transaction. The CSP specification for *PROP* has already been given in Section 2. This property can now be checked for trace refinement against the full specification (with events not occurring in *PROP* hidden), i.e. using FDR2 syntax we check

```
assert PROP [T= SYSTEM \ {|request, execute, inform,
                           decide, result, acknowledge|}
```

We ran FDR2 on a Linux PC (Open SUSE 10.2) equipped with a 3 GHz Pentium 4 processor and 1 GB RAM. In Table 1, we give an overview of the computation times and sizes of the generated state spaces for using FDR2 to check *PROP*

- directly calling FDR2 on *System* without using compositional reasoning,
- using L* and assume-guarantee reasoning based on the *given* decomposition into *Coord* and *Pages*,
- using L* and assume-guarantee reasoning based on *our* decomposition based on three different sequential cuts:
  1. $Vot_1 \| Vot_2$ for the cut $\{vote\}$,
  2. $Dec_1 \| Dec_2$ for the cut $\{decide, vote\}$,
  3. $Inf_1 \| Inf_2$ for the cut $\{inform\}$.

We started with one instance of the component *Page* and incrementally increased N – the value used is given in the third column. The fourth column displays the verification time in seconds; column 5 and 6 indicate the size of the computed state space for components 1 and 2, respectively. One asterisk symbolizes that the machine ran out of memory due to exceeding its swap limit after approximately 90 minutes whereas two asterisks denote that there was no computation result after more than four hours.

Apparently, if we use our decomposition and L*, we are able to verify the property for a higher N compared to verification of the original specification. Furthermore, we achieve much better runtime results. The best results are achieved for the decomposition based on the cut $\{decide, vote\}$. Since the cut $\{decide, vote\}$ outperforms the cut $\{vote\}$, we can deduce that the smallest cut may not always achieve the best results. In particular, in our example, $\{vote\}$ leads to $V_{\mathbf{C}} = \{votes\}$ – this variable is not represented in the corresponding set for $\{decide, vote\}$ since we are decomposing the specification at a point after which *votes* will never be used again.

Despite this there is no significant difference between the verification times for the different cuts we were looking at. The generated assumptions for N equals 2 for two of the cuts are depicted in Figure 3 (omitted from address parameters;

Table 1. Experimental Results for FDR2

System	L*	N	Time/sec	States Comp.1	States Comp.2
$Coord \mid\mid Pages$	no	1	<1	Full System: 47	
		2	<1	Full System: 1116	
		3	<1	Full System: 26190	
		4	6	Full System: 623376	
		5	227	Full System: 14984838	
		6	(*)	Full System: unknown	
$Coord \mid\mid Pages$	yes	1	1	9	65
		2	9	17	4353
		3	35	24	287496
		4	656	31	18974736
		5	(*)	37	(**)
$Vot_1 \mid\mid Vot_2$	yes	1	<1	17	12
		2	1	288	53
		3	3	4374	217
		4	5	64800	893
		5	21	949158	3673
		6	302	13845168	15053
		7	(*)	(**)	61417
$Dec_1 \mid\mid Dec_2$	yes	1	<1	23	11
		2	1	324	50
		3	2	4590	210
		4	5	66096	878
		5	19	956934	3642
		6	236	13891824	14990
		7	(*)	(**)	61290
$Inf_1 \mid\mid Inf_2$	yes	1	<1	29	8
		2	1	432	77
		3	2	6102	639
		4	4	85536	5201
		5	19	1197990	41799
		6	275	16831152	333137
		7	(*)	(**)	2640759

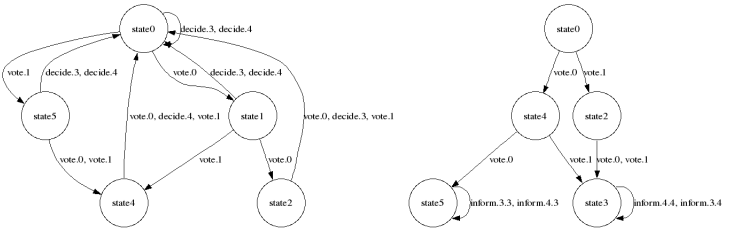


Fig. 3. Final assumptions for *PROP* based on Cuts $\{decide, vote\}$ and $\{inform\}$

0, 1 and 3, 4 are abstractions of *YES*, *NO* and *COMMIT*, *ABORT*, respectively). Runtime behaviour is worst for the given decomposition. Here, the results suffer from a larger generated assumption with more states and transitions.

5 Conclusion and Related Work

This paper presented an approach to compositional verification illustrated on specifications written in the integrated formal method CSP-OZ. We decomposed

a specification such that the resulting components can efficiently be used for assume-guarantee reasoning. The decomposition is performed on the specification's dependence graph. We showed correctness of the approach. In the context of automatically generating assumptions we illustrated the technique on an example specification. Verification results are carried out using the CSP model checker FDR2.

Related work. Assume-guarantee reasoning was first introduced by Chandy and Misra [19] and Jones [17, 16]. Compositional verification for integrated formal methods undergoes intensive research. For CSP‖B, a coupling of the B method with CSP, Treharne and Schneider explored compositional proof techniques [24] by also using FDR2 for verification of CSP processes.

The static analysis of a specification is the foundation for the decomposition technique proposed in this paper. Brueckner [4] defined a CSP-OZ dependence graph which we incorporated here. He used it to compute a specification slice [26] with respect to a certain property. Our decomposition technique is more closely related to the technique of program *chopping* [22] – we do not compute full specification slices but rather chop the specification up to the point at which the assumption holds.

Cobleigh et al. first used the L* algorithm in the context of automatic learning an assumption for compositional reasoning [8] in the domain of Labelled Transition Systems. We implemented their framework in our context for FDR2.

Alur and Nam [20] do assume-guarantee based reasoning in the context of symbolic model checking. They also use L* to automatically generate assumptions and decompose a given system. The decomposition is computed fully automatically in terms of hypergraph partitioning. In their semantic domain of *symbolic transition modules* based on solely boolean variables, they do not deal with control flow, synchronisation and dependence graphs.

Future work. This paper is intended to provide the basic concept for a decomposition and automated verification technique for CSP-OZ. There are many follow-up steps to be taken, some of which are described below.

The requirement that the dependence graph is sequential is quite strong. To relax this restriction, we need to deal with circularity within the DG. The technique proposed in this paper will thus be extended to specifications with a recursive structure. In this case, we aim to reuse the decomposition technique by defining *two* cuts determining the switch from the first to the second phase and vice versa. This will lead to the application of a symmetric assume-guarantee proof rule where two assumptions can be learned simultaneously [2].

Even though most of the steps within this framework can be performed automatically, such as the computation of the DG, the translation of a CSP-OZ class to the input language of FDR2 and the assumption learning, the definition of a cut is currently done by hand. To find an optimal cut in the sense of evenly distributed components we might use techniques presented in [14, 27] to compare several possible decompositions in terms of a *lattice of decomposition slices*. Other heuristics need to be defined and evaluated.

Finally, we aim at evaluating the approach on a much bigger case study.

Acknowledgement. We thank Ramsay Taylor for fruitful discussions on the topic and for correcting our English.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75, 87–106 (1987)
2. Barringer, H., Giannakopoulou, D., Pasareanu, C.S.: Proof rules for automated compositional verification through learning. In: International Workshop on Specification and Verification of Component Based Systems, Finland (2003)
3. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison (1987)
4. Brückner, I.: Slicing Integrated Formal Specifications for Verification. PhD thesis, Universität Paderborn (2008)
5. Brückner, I., Wehrheim, H.: Slicing an integrated formal method for verification. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 360–374. Springer, Heidelberg (2005)
6. Clarke, E., Emerson, E., Sistla, A.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2), 244–263 (1986)
7. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In: ISSTA 2006: Proceedings of the 2006 international symposium on Software testing and analysis, pp. 97–108. ACM Press, New York (2006)
8. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
9. de Roever, W.P., Hanneman, U., Hooiman, J., Lakhneche, Y., Poel, M., Zwiers, J., de Boer, F.: Concurrency Verification. Cambridge University Press, Cambridge (2001)
10. Elrad, T., Francez, N.: Decomposition of distributed programs into communication-closed layers. Sci. Comput. Program. 2(3), 155–173 (1982)
11. Fischer, C.: CSP-OZ: A combination of Object-Z and CSP. In: Formal Methods for Open Object-Based Distributed Systems (FMOODS 1997), vol. 2, pp. 423–438. Chapman and Hall, Boca Raton (1997)
12. Fischer, C., Wehrheim, H.: Model-checking CSP-OZ specifications with FDR. In: IFM, pp. 315–334 (1999)
13. Francez, N., Pnueli, A.: A proof method for cyclic programs. Acta Informatica 9(2) (1978)
14. Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17(8), 751–761 (1991)
15. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
16. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
17. Jones, C.B.: Tentative steps towards a development method for interfering programs. Transactions on Programming Languages and Systems 5(4), 596–619 (1983)

18. Formal Systems (Europe) Ltd. Failure divergence refinement: Fdr2 user manual (1997)
19. Misra, J., Chandy, K.M.: Proofs of networks of processes. IEEE Trans. Softw. Eng. 7(4), 417–426 (1981)
20. Nam, W., Alur, R.: Learning-based symbolic assume-guarantee reasoning with automatic decomposition. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 170–185. Springer, Heidelberg (2006)
21. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)
22. Reps, T.W., Rosay, G.: Precise interprocedural chopping. In: SIGSOFT FSE, pp. 41–52 (1995)
23. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River (1997)
24. Schneider, S., Treharne, H.: Verifying controlled components. In: IFM, pp. 87–107 (2004)
25. Smith, G.: The Object-Z Specification Language. Kluwer Academic Publishers, Dordrecht (2000)
26. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
27. Tonella, P.: Using a concept lattice of decomposition slices for program understanding and impact analysis. IEEE Trans. Software Eng. 29(6), 495–509 (2003)
28. Weiser, M.: Programmers use slices when debugging. Commun. ACM 25(7), 446–452 (1982)
29. Wonisch, D.: Automatisiertes kompositionelles Model Checking von CSP Spezifikationen. Bachelor's thesis, Universität Paderborn (April 2008)

A Formal Soundness Proof of Region-Based Memory Management for Object-Oriented Paradigm*

Florin Craciun[1], Shengchao Qin[1], and Wei-Ngan Chin[2]

[1] Department of Computer Science, Durham University, UK
{florin.craciun,shengchao.qin}@durham.ac.uk
[2] Department of Computer Science, National University of Singapore, Singapore
chinwn@comp.nus.edu.sg

Abstract. Region-based memory management has been proposed as a viable alternative to garbage collection for real-time applications and embedded software. In our previous work we have developed a region type inference algorithm that provides an automatic compile-time region-based memory management for object-oriented paradigm. In this work we present a formal soundness proof of the region type system that is the target of our region inference. More precisely, we prove that the object-oriented programs accepted by our region type system achieve region-based memory management in a safe way. That means, the regions follow a stack-of-regions discipline and regions deallocation never create dangling references in the store and on the program stack. Our contribution is to provide a simple syntactic proof that is based on induction and follows the standard steps of a type safety proof. In contrast the previous safety proofs provided for other region type systems employ quite elaborate techniques.

1 Introduction

Modern object-oriented programming languages provide a run-time system that automatically reclaims memory using tracing garbage collection [24]. A correct garbage collector can guarantee that the memory is not collecting too early, and also that all memory is eventually reclaimed if the program terminates. However the space and time requirements of garbage-collected programs are very difficult to estimate in practice. Therefore many different solutions have been proposed for real-time applications and embedded software running on resource-limited platforms. These solutions either completely omit the use of garbage collectors (e.g. JavaCard platform), or use real-time garbage collectors [1], or use region-based memory management (e.g. Real-Time Specification for Java (RTSJ) [3]).

Region-based memory management systems allocate each new object into a program-specified *region*, with the entire set of objects in each region deallocated simultaneously when the region is deleted. Various studies have shown that region-based memory management can provide memory management with good real-time performance. Individual object deallocation is accurate but time unpredictable, while region deletion presents a better temporal behavior, at the cost of some space overhead. Data

* The work is supported in part by the EPSRC project EP/E021948/1.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 126–146, 2008.

locality may also improve when related objects are placed together in the same region. Classifying objects into regions based on their lifetimes may deliver better memory utilization if regions are deleted in a timely manner.

The first safe region-based memory system was introduced by Tofte and Talpin [22,23] for a functional language. Using a region type inference system, they have provided an automatic static region-based memory management for Standard ML. More precisely, their compiler can group heap allocations into regions and it can statically determine the program points where it is safe to deallocate the regions. Later, several projects have investigated the use of region-based memory management for C-like languages (e.g. Cyclone [13]) and object-oriented languages [9,5]. These projects provide region type checkers and require programmers to annotate their programs with region declarations. The type checkers then use these declarations to verify that well-typed programs safely use the region-based memory.

In our previous work [8], we have developed the first automatic region type inference system for object-oriented paradigm. Our compiler automatically augments unannotated object-oriented programs with regions type declarations and inserts region allocation/deallocation instructions that achieve a safe memory management. In this paper we provide the safety proof of our region type system that is the target of our previous region inference algorithm.

In our work, we use *lexically-scoped regions* such that the memory is organised as a *stack of regions*, as illustrated in Fig. 1. Regions are memory blocks that are allocated and deallocated by the construct `letreg r in e`, where the region `r` can only be used to allocate objects in the program `e`. The older regions (with longer lifetime) are allocated at the bottom of the stack while the younger regions (with shorter lifetime) are at the top. The region lifetime relations are expressed using a transitive *outlive relation*, denoted by $\succeq$. Thus, we can define the lifetime constraints $\texttt{r0}\succeq\texttt{r1}\wedge\texttt{r1}\succeq\texttt{r2}\wedge\texttt{r2}\succeq\texttt{r3}\wedge\texttt{r3}\succeq\texttt{r4}$ on the regions of Fig. 1. Region lifetime constraints (as shown in Fig. 2) are of two main forms $r_1 \succeq r_2$ and $r_1 = r_2$. The constraint $r_1 \succeq r_2$ indicates that the lifetime of region r_1 is not shorter than that of r_2, while the constraint $r_1 = r_2$ denotes that r_1 and r_2 must be the same region. The equality can be expressed as an outlive relation such that $r_1 = r_2$ iff $r_1 \succeq r_2$ and $r_2 \succeq r_1$.

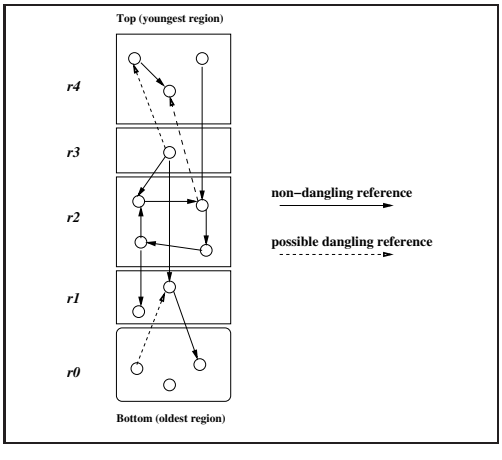


Fig. 1. Lexically-Scoped Regions

Dangling references are a safety issue for region-based memory management. Fig. 1 shows two kinds of references: non-dangling references and possible dangling references. Non-dangling references originate from objects placed in a younger region and

point to objects placed either in an older region or inside the same region. Possible dangling references occur when objects placed in an older region point to objects placed in a younger region. They turn into dangling references when the younger region is deallocated. Using a dangling reference to access memory is unsafe because the accessed memory may have been recycled to store other objects. There are two approaches to eliminating this problem. The first approach allows the program to create dangling references, but uses an effect-based region type system to ensure that the program never accesses memory through a dangling reference [22,23,9,13]. The second approach uses a region type system to prevent the program from creating dangling references at all [5]. Our work has adopted the second approach.

Contributions. The main contribution of this paper is the soundness proof of our region type system for object-oriented paradigm. We prove that our region type system guarantees that well-typed programs use lexically-scoped regions and never create dangling references in the store and on the program stack. We provide a simple syntactic proof based on induction (rather than a more elaborate co-induction machinery), that follows the standard steps of a type safety proof [25]. Our small-step dynamic semantics decomposes high-level expression `letreg r in e` into three intermediate operations: allocation of region `r` on the stack, evaluation of program `e`, and deallocation of region `r`. The difficulty is to prove that after deallocation of region `r`, the store, the program stack and the remaining code do not contain any reference to region `r` and to the objects stored in region `r`. To prove that region deallocation is safe, we use the region constraints of our type system and a syntactic condition that we imposed to restrict the valid intermediate code. However our syntactic restriction does not restrict high-level source code, it only defines the correct intermediate code to which high-level code can be evaluated.

Related Work. In the original effect-based region type system, Tofte and Talpin [23,21,2] and later Christiansen and Velschow [9], in their region calculus for object-oriented languages make use of co-induction to prove the soundness. Their proof requires co-induction partly because they prove two properties at the same time: type soundness and translation soundness. The latter property guarantees that there exists a semantic relation between source program and its region-annotated counterpart. Our safety theorems are only focused on the problem of type soundness, thus are simpler to prove. A co-inductive definition is required in their proof also because they use a big-step semantics where certain information is lost when deleting a region from the store, as discussed in [15,7]. Our system uses a small-step operational semantics instrumented with regions which makes the consistency definition and the proof easier. Calcagno [6] uses a stratified operational semantics to avoid co-induction in the proof of safety properties of a simple version of Tofte and Talpin's region calculus, while Helsen et al. [15,14] introduces a special constant for defunct regions in their big-step semantics which makes the soundness proof simpler. A similar proof with ours is the safety proof of Niss [19], that in addition to a simple functional language handles an imperative calculus, and like our proof avoids explicit co-induction by using store typing. Cyclone [13] also has an effect system used for a soundness proof and does not use co-induction. Elsman [12] refines Tofte and Talpin's region type system in order to forbid the dangling references and proves by induction the safety for a small functional

language. There are many differences between his proof and ours. His proof is based on a small-step contextual semantics [17], while in our proof we explicitly model the heap as a stack of regions and we use a consistency relation between the static and dynamic semantics. In addition Elsman uses a syntax-directed containment relation to express the regions of the program values and also to force the stack discipline for regions' allocation and deallocation. In our case the region requirements and the order among regions are expressed by the region constraints of the type system. However we also impose a syntactic condition to restrict the valid intermediate (non-source) programs. Boudol [4] refines Tofte and Talpin's region calculus to a flow-sensitive effect-based region type system, that explicitly records the deallocations effects. He provides a simple proof for a functional language by means of a subject reduction property up to simulation. Although his simulation is half-bisimulation, his proof does not employ co-induction. In contrast our region type system is a flow-insensitive calculus. However our syntactic restriction on intermediate code has a similar role as the flow-sensitive deallocation effect. Our type system is similar to SafeJava's type system of Boyapati et al. [5], but in addition we support the region subtyping principle [13]. However SafeJava does not provide a formal proof for its region type system.

Outline. The paper is organized as follows. Section 2 introduces the syntax of our region calculus. Section 3 presents our region type system, while Section 4 defines the dynamic semantics of our region calculus. Section 5 extends the static semantics to intermediate expressions, while Section 6 presents the soundness theorems. A brief conclusion is given. The technical report [11] contains the details of our inductive proofs.

2 Region Calculus

Our region calculus is designed by annotating with regions a Java-like object-oriented language, named Core-Java [10]. The full syntax of the region-annotated Core-Java language is given in Fig. 2. Core-Java is designed in the same minimalist spirit as the pure functional calculus Featherweight Java [16]. Despite its expression-oriented syntax, Core-Java supports imperative features.

Each class definition is parameterized with one or more regions to form a *region type*. For instance, a region type $cn\langle r_1, ..., r_n\rangle$ is a class name cn annotated with region parameters $r_1...r_n$. Parameterization allows us to obtain a region-polymorphic type for each class whose fields can be allocated in different regions. The first region parameter r_1 is special: it refers to the region in which the instance object of this class is allocated. The fields of the objects, if any, are allocated in the other regions $r_2...r_n$ which should *outlive* the region of the object. This is expressed by the constraint $\bigwedge_{i=2}^{n}(r_i \succeq r_1)$, which captures the property that the regions of the fields (in $r_2...r_n$) should have lifetimes no shorter than the lifetime of the region (namely r_1) of the object that refers to them. This condition, called *no-dangling requirement*, prevents dangling references completely, as it guarantees that each object never references another object in a younger region. In general the class invariant, φ, of a class consists of the no-dangling requirement for the region type of the current class, the no-dangling requirements for the fields' region types, and the class invariant of the parent class We do not require region parameters

$$
\begin{array}{lll}
t & ::= cn\langle r^+\rangle \mid prim\langle\rangle \mid \bot & (\textbf{region types}) \\
prim & ::= \textbf{int} \mid \textbf{boolean} \mid \textbf{void} & \\
\varphi & ::= r_1 \succeq r_2 \mid r_1 = r_2 \mid true \mid \varphi_1 \wedge \varphi_2 & (\textbf{region constraints}) \\
P & ::= def^* & (\textbf{region annotated program}) \\
def & ::= \textbf{class}\ cn_1\langle r^+\rangle\ \textbf{extends}\ cn_2\langle r^+\rangle\ \textbf{where}\ \varphi & \\
 & \quad \{(t\,f)^*\ meth^*\} & (\textbf{region annotated class declaration}) \\
meth & ::= t\ mn\langle r^*\rangle((t\ v)^*)\ \textbf{where}\ \varphi\ \{e\} & (\textbf{region annotated method}) \\
e & ::= \textbf{null} \mid k \mid v \mid v.f \mid v = e \mid v.f = e & (\textbf{region annotated expression}) \\
 & \mid e_1\ ;\ e_2 \mid \{(t\ v)\ e\} \mid \textbf{new}\ cn\langle r^+\rangle(v^*) & \\
 & \mid v.mn\langle r^*\rangle(v^*) \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid \textbf{while}\ v\ e & \\
 & \mid \textbf{letreg}\ r\ \textbf{in}\ e & (\textbf{region declaration}) \\
 & cn \in \textit{class names} & r \in \textit{region variable names} \\
 & mn \in \textit{method names} & k \in \textit{integer or boolean constants} \\
 & f \in \textit{field names} & v \in \textit{variable names}
\end{array}
$$

Fig. 2. The Syntax of Region-Annotated Core-Java

for primitive types, since primitive values can be copied and stored directly on the stack or they are part of an object. In order to keep the same notation, we use $prim\langle\rangle$ to denote a region annotated primitive type. Although null values are of object type, they are regarded as primitive values. The type of a null value is denoted by $\bot$.

The *region subtyping principle* allows an object from a region with longer lifetime to be assigned to a location where a region with a shorter lifetime is expected. This principle is illustrated by the subtyping rule [RegSub] of Fig. 3. This rule relies on the fact that once an object is allocated in a particular region, it stays within the same region and never migrates to another region. This property allows us to apply covariant subtyping to the region of the current object. However, the object fields are mutable (in general) and must therefore use invariant subtyping to ensure the soundness of subsumption. The other two rules, [SubClass] and [Null] from Fig. 3 denote the class subtyping and the fact that a null value can be assigned to any object, respectively.

$$
\frac{\varphi{=}(x_1{\succeq}\hat{x}_1) \wedge \bigwedge_{i=2}^{n}(x_i{=}\hat{x}_i)}{\vdash cn\langle x_{1..n}\rangle{<:}cn\langle \hat{x}_{1..n}\rangle,\ \varphi}\ [\textbf{RegSub}]
$$

$$
\frac{\textbf{class}\ cn\langle r_{1..n}\rangle\ \textbf{extends}\ cn'\langle r_{1..m}\rangle.. \in P' \quad n{\geq}m{\geq}p \quad \vdash cn'\langle x_{1..m}\rangle{<:}cn''\langle x'_{1..p}\rangle,\ \varphi}{\vdash cn\langle x_{1..n}\rangle{<:}cn''\langle x'_{1..p}\rangle,\ \varphi}\ [\textbf{SubClass}]
$$

$$
\frac{}{\vdash \bot{<:}cn\langle x_{1..n}\rangle,\ true}\ [\textbf{Null}]
$$

Fig. 3. Region Subtyping Rules

Every method is decorated with zero or more region parameters; these parameters capture the regions used by each method's parameters (including `this`) and result. For simplicity, no other externally defined regions are made available for a method. Thus, all regions used in a method either are mapped to these region parameters or are localised by `letreg` in the method body. Each method also has a method precondition, φ

expressed as a region lifetime constraint that is consistent with the operations performed in the method body. The method precondition also contains the class invariants of its parameters including the receiver and its result. The instance methods of a subclass can override the instance methods of the superclass.

Consider the `Pair` class in Fig. 4. As there are two fields in this class, a distinct region is introduced for each of them, `r2` for `fst` field and `r3` for `snd` field. The `Pair` object is placed in the region `r1`. To ensure that every `Pair` instance satisfies the no-dangling requirement, the region lifetime constraint `r2≥r1∧r3≥r1` is added to the class invariant.

```
class Pair⟨r1,r2,r3⟩ extends Object⟨r1⟩
   where r2≥r1 ∧ r3≥r1 {
   Object⟨r2⟩ fst;
   Object⟨r3⟩ snd;

void setSnd⟨r1,r2,r3,r4⟩(Object⟨r4⟩ o)
   where r4≥r3∧r2≥r1∧r3≥r1
    {snd=o;}
void swap⟨r1,r2,r3⟩() where r2=r3∧r2≥r1
   { Object⟨r2⟩ tmp=fst;fst=snd;snd=tmp}
Pair⟨r5,r6,r7⟩ exalloc⟨r1,r2,r3,r5,r6,r7⟩()
    where r7≥r5∧r6≥r5∧r2≥r1∧r3≥r1
  {letreg r in {
    Pair⟨r7,r7,r7⟩ p4;
    Pair⟨r,r,r⟩ p3;
    Pair⟨r5,r6,r7⟩ p2;
    Pair⟨r,r,r⟩ p1;
    p4 = new Pair⟨r7,r7,r7⟩(null,null);
    p3 = new Pair⟨r,r,r⟩(p4,null);
    p2 = new Pair⟨r5,r6,r7⟩(null,p4);
    p1 = new Pair⟨r,r,r⟩(p2,null);
    p1.setSnd⟨r,r,r,r⟩(p3); p2} }
}
```

Fig. 4. Region-Annotated Core-Java Program

Consider the `setSnd`, `swap`, and `exalloc` methods of the `Pair` class. A set of distinct region parameters are introduced for the methods' parameters, and the results, as shown in Fig. 4. The receiver regions are taken from the class definition. Moreover, the methods' region lifetime constraints are based on the possible operations of the respective methods. For example, due to an assignment operation and region subtyping, we have `r4≥r3` for `setSnd`, while `r2=r3` is present due to the swapping operation on the receiver object in the `swap` method. Though the `swap` method's region constraint is exclusively on the regions of the current object, we associate the constraint with the method. In this way, only those objects that might call the method are required to satisfy this constraint. The class invariants of methods' parameters (including the receiver and their result) are also added to the methods' region constraints. The `exalloc` method's body introduces a local region `r` using `letreg`. Since the `p1` and `p3` objects do not escape from the `exalloc` method's body, they are stored in the local region `r`. The `p2` and `p4` objects escape through the method result, therefore they are stored in the method result's regions `r5` and `r7`, respectively.

$$\frac{WFClasses(P) \quad P = def_1\ ..\ def_n \quad FieldsOnce(def_i)\ \ i = 1..n \quad MethodsOnce(def_i)\ \ i = 1..n \quad P \vdash InheritanceOK(def_i)\ \ i = 1..n \quad P \vdash_{def} def_i\ \ i = 1..n}{\vdash P}\ [\text{RC-PROG}]$$

$$\frac{def = \textbf{class}\ cn\langle r_{1..n}\rangle \textbf{extends}\ c\langle r_{1..m}\rangle \ \textbf{where}\ \varphi\ \{field_{1..p}\ meth_{1..q}\} \quad r_1 \notin \bigcup_{i=1}^{p} reg(field_i) \quad \varphi \Rightarrow r_i \succeq r_1\ \ i = 2..n \quad R = \{r_1, \ldots, r_n\} \quad P; \{this : cn\langle r_{1..n}\rangle\}; R; \varphi \vdash_{meth} meth_i\ \ i = 1..q \quad P; R; \varphi \vdash_{field} field_i\ \ i = 1..p}{P \vdash_{def} def}\ [\text{RC-CLASS}]$$

$$\frac{\Gamma' = \Gamma + (v_j : t_j)_{j:1..p} \quad R' = R \cup \{r_1, \ldots, r_m\} \quad \varphi' = \varphi \wedge \varphi_0 \quad P; R'; \varphi' \vdash_{type} t_j,\ j = 0..p \quad P; \Gamma'; R'; \varphi' \vdash e : t_0 \quad P; R'; \varphi' \vdash t_0 <: t_0}{P; \Gamma; R; \varphi \vdash_{meth} t_0\ mn\langle r_{1..m}\rangle((t_j\ v_j)_{j:1..p}) \textbf{where}\ \varphi_0\ \{e\}}\ [\text{RC-METH}]$$

$$\frac{P; R; \varphi \vdash_{type} t' \quad \Gamma' = \Gamma + (v : t') \quad P; \Gamma'; R; \varphi \vdash e : t}{P; \Gamma; R; \varphi \vdash \{(t'\ v)\ e\} : t}\ [\text{RC-EB}]$$

$$\frac{(v : t) \in \Gamma}{P; \Gamma; R; \varphi \vdash v : t}\ [\text{RC-VAR}]$$

$$\frac{P; R; \varphi \vdash_{type} cn\langle r_{1..n}\rangle \quad fieldlist(cn\langle r_{1..n}\rangle) = (t_i\ f_i)_{i:1..p} \quad (v_i : t'_i) \in \Gamma \quad P; R; \varphi \vdash t'_i <: t_i \quad i = 1..p}{P; \Gamma; R; \varphi \vdash \textbf{new}\ cn\langle r_{1..n}\rangle(v_1, .., v_p) : cn\langle r_{1..n}\rangle}\ [\text{RC-NEW}]$$

$$\frac{(v_0 : cn\langle a^+\rangle) \in \Gamma \quad P; R; \varphi \vdash_{type} cn\langle a^+\rangle \quad (t\ mn\langle a^+ r'^+\rangle((t_i\ v_i)_{i:1..n}) \textbf{where}\ \varphi_0\ \{e\}) \in cn\langle a^+\rangle \quad (v'_i : t'_i)_{i:1..n} \in \Gamma \quad a'^+ \in R \quad \rho = [r'^+ \mapsto a'^+] \quad \varphi \Rightarrow \rho\,\varphi_0 \quad P; R; \varphi \vdash t'_i <: \rho\, t_i \quad i = 1..n}{P; \Gamma; R; \varphi \vdash v_0.mn\langle a^+ a'^+\rangle(v'_1..v'_n) : \rho\, t}\ [\text{RC-INVOKE}]$$

$$\frac{a = fresh() \quad \varphi' = \varphi \wedge \bigwedge_{r' \in R}(r' \succeq a) \quad P; \Gamma; R \cup \{a\}; \varphi' \vdash [r \mapsto a]e : t \quad reg(t) \subseteq R}{P; \Gamma; R; \varphi \vdash \textbf{letreg}\ r\ \textbf{in}\ e : t}\ [\text{RC-LETR}]$$

$\rho t,\ \rho\varphi,\ \rho e$ region substitution on a type, a constraint, and an expression
$fresh()$ returns one or more new/unused region names

Fig. 5. Region Type Checking Rules

3 Region Type System: Static Semantics

Our region type system guarantees that region-annotated Core-Java programs never create dangling references. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase. A part of region type checking rules are depicted in Fig. 5, with some auxiliary rules in Fig. 6 (a complete description of region type system is given in [11]). Judgments of the following forms are employed:

- $\vdash P$ denoting that a program P is well-typed.
- $P \vdash_{def} def$ denoting that a class declaration def is well-formed.
- $P;\Gamma;R;\varphi \vdash_{meth} meth$ denoting that a method $meth$ is well-defined with respect to the program P, the type environment Γ, the set of live regions R, and the region constraint φ.
- $P;\Gamma;R;\varphi \vdash e{:}t$ denoting that an expression e is well-typed with respect to the program P, the type environment Γ, the set of live regions R, and the region constraint φ.

- $P; R; \varphi \vdash_{type} t$ denoting that a type t is well-formed, namely, the regions of the type t are from the set of the live regions R, and the invariant of the type t is satisfied by the constraint context φ.
- $P; R \vdash_{constr} t,\ \varphi$ denoting that the regions of the type t are from the set of the live regions R, while φ is the invariant of the type t.
- $P; R; \varphi \vdash_{field} field$ denoting that the type of a field *field* is well-formed with respect to $\vdash_{type}$ judgment.
- $P; R; \varphi \vdash t <: t'$ denoting that the type t is a subtype of the type t', namely both types are well-formed and the region constraint of the subtyping relation (defined in Fig. 3) is satisfied by the constraint context φ.

The rule [RC–PROG] denotes that a region-annotated program is well-typed if all declared classes are well-typed. The predicates in the premise are used to capture the standard well-formedness conditions for the object-oriented programs such as no duplicate definitions of classes and no cycle in the class hierarchy; no duplicate definitions of fields; no duplicate definitions of methods; and soundness of class subtyping and method overriding.

The rule [RC–CLASS] indicates that a class is well-formed if all its fields and methods are well-formed, and the class invariant ensures the necessary lifetime relations among class region parameters. In addition, the rule does not allow the first region of the class to be used by the region types of the fields. Using the first region on a field would break the object (region) subtyping (rule [RegSub] of Fig. 3). Function $reg(field_i)$ returns the region variables of a field type (see Fig. 6).

$$reg(\{\}) =_{def} \{\} \quad reg(\{v{:}\tau\langle r^*\rangle\} \cup \Gamma) =_{def} \{r^*\} \cup reg(\Gamma)$$

$$reg(\tau\langle r^*\rangle) =_{def} \{r^*\} \quad reg((\tau\langle r^*\rangle\ f)) =_{def} \{r^*\}$$

$$reg(r_1 \succeq r_2) =_{def} \{r_1, r_2\} \quad reg(r_1 = r_2) =_{def} \{r_1, r_2\}$$

$$reg(true) =_{def} \{\} \quad reg(\varphi_1 \wedge \varphi_2) =_{def} reg(\varphi_1) \cup reg(\varphi_2)$$

$$fieldlist(\mathbf{Object}\langle r\rangle) =_{def} [\,]$$

$$\frac{\mathbf{class}\ cn_1\langle r_{1..n}\rangle\ \mathbf{extends}\ cn_2\langle r_{1..m}\rangle .. \{(t_i\ f_i)_{i:1..p} ..\} \in P' \quad \ell = fieldlist(\rho\ cn_2\langle r_{1..m}\rangle) \quad \rho = [r_i \mapsto x_i]_{i=1}^{n}}{fieldlist(cn_1\langle x_{1..n}\rangle) =_{def} \ell +\!\!+ [(\rho\ t_i)\ f_i]_{i=1}^{p}}$$

Fig. 6. Auxiliary Region Checking Rules

The rule [RC–METH] checks the well-formedness of a method declaration. Each region type is checked to be well-formed, that means its regions are in the current set of live regions and its invariant is satisfied by the current constraint context. The method body is checked using the type relation for expressions such that the gathered type has to be a subtype of the declared type.

Our type relation for expressions is defined in a syntax-directed fashion. Take note that region constraints of the variables are not checked at their uses ([RC–VAR]), but at their declaration sites ([RC–EB]). The region invariant of an object is also checked when that object is created ([RC–NEW]). In the rule for object creation ([RC–NEW]), the function $fieldlist(cn\langle x_{1..n}\rangle)$ returns a list comprising all declared and inherited fields of the class $cn\langle x_{1..n}\rangle$ and their region types according to the regions $x_1..x_n$ of the class cn (see Fig. 6). They are organized in an order determined by the constructor function.

The rule [RC–INVOKE] is used to check a method call. It ensures that the method region parameters are live regions and the method precondition is satisfied by the current constraint context as $\varphi \Rightarrow \rho\varphi_0$. A substitution ρ is computed for the method's formal region parameters. The current arguments are also checked to be subtypes of the method's formal parameters.

The rule [RC–LETR] is used to check a local region declaration. The local expression is checked with an extra live region a (that is a fresh region), and an extra constraint $\bigwedge_{r' \in R}(r' \succeq a)$ that ensures that newly introduced region is on the top of the region stack. The rule uses a region substitution on the expressions. Note that the region substitutions on expressions, constraints and types are defined as expected. The gathered region type of the local expression is checked to contain only live regions (from R excepting a). This guarantees that the localized region a does not escape. Function $reg(t)$ returns all region variables of t (see Fig. 6).

4 Dynamic Semantics

In this section we define the dynamic semantics of our region calculus. Our dynamic semantics rules use runtime checks to throw an error and to abort the execution, whenever the evaluation of a region-annotated Core-Java program tries to create a dangling reference. In Section 6 we prove that those runtime checks are redundant for well-typed programs, namely the evaluation of a well-typed region-annotated Core-Java program never creates a dangling reference. The dynamic semantics is defined as a small-step rewriting relation from machine states to machine states. A machine state is of the form $\langle \varpi, \Pi \rangle [e]$, where ϖ is the heap organized as a stack of regions, Π is the variable environment, and e is the current program. Our dynamic semantics was inspired by the previous work on abstract models of memory management [18] and region-based memory management [9,13]. The following notations are used:

Region Variables :	$r, a \in RegVar$
Offset :	$o \in Offset$
Locations :	ℓ *or* $(r, o) \in Location = RegVar \times Offset$
Primitive Values :	$k \mid \texttt{null} \in Prim$
Values :	$\delta \in Value = Prim \uplus Location$
Variable Environment :	$\Pi \in VEnv = Var \rightharpoonup_{\text{fin}} Value$
Field Environment :	$V \in FEnv = FieldName \rightharpoonup_{\text{fin}} Value$
Object Values :	$cn\langle r^* \rangle (V) \in ObjVal = ClassName \times (RegVar)^n \times FEnv$
Store :	$\varpi \in Store = [\,] \mid [r \mapsto Rgn] Store$
Runtime Regions :	$Rgn \in Region = Offset \rightharpoonup_{\text{fin}} ObjVal$

Regions are identified by region variables. We assume a denumerably infinite set of region variables, *RegVar*. The store ϖ is organized as a stack, that defines an ordered map from region variables, r to runtime regions *Rgn*. The notation $[r \mapsto Rgn]\varpi$ denotes a stack with the region r on the top, while $[\,]$ denotes an empty store. The store can only be extended with new region variables. A runtime region *Rgn* is an unordered finite map

from offsets to object values. We assume a denumerably infinite set of offsets, *Offset* for each runtime region *Rgn*.

The set of values that can be assigned to variables and fields is denoted by *Value*. Such a value is either a primitive value (a constant or a null value) or it is a location in the store. A location consists of a pair of a region variable and an offset.

An object value consists of a region type $cn\langle r^* \rangle$, and a field environment V mapping field names to values. V is not really an environment since it can only be updated, never extended. An update of field f with value δ is written as $V+\{f \mapsto \delta\}$.

The variable environment Π is a mapping $Var \rightharpoonup_{\text{fin}} Value$, while the type environment Γ that corresponds to the runtime variable environment is also a mapping $Var \rightharpoonup_{\text{fin}} Type$. To avoid variable name duplication, we assume that the local variables of the blocks and the arguments of the functions are uniquely renamed in a preprocessing phase.

Notation $f : A \rightharpoonup_{\text{fin}} B$ denotes a partial function from A to B with a finite domain, written $A = dom(f)$. We write $f+\{a \mapsto b\}$ for the function like f but mapping a to b (if $a \in dom(f)$ and $f(a)=c$ then $(f+\{a \mapsto b\})(a)=b$). The notation $\{\}$ (or $\emptyset$) stands for an undefined function. Given a function $f : A \rightharpoonup_{\text{fin}} B$, the notation $f-C$ denotes the function $f_1 : (A-C) \rightharpoonup_{\text{fin}} B$ such that $\forall x \in (A-C) \cdot f_1(x)=f(x)$.

We require some intermediate expressions for the small-step dynamic semantics to follow through. The intermediate expressions help our proof to use simpler induction techniques rather than a more elaborate co-induction machinery. The syntax of intermediate expressions is thus extended from the original expression syntax, as follows:

$$e ::= \ldots \mid (r, o) \mid \texttt{ret}(v, e) \mid \texttt{retr}(r, e)$$

The expression $\texttt{ret}(v, e)$ is used to capture the result of evaluating a local block, or the result of a method invocation. The variable associated with `ret` denotes either a block local variable or a method receiver or a method parameter. This variable is popped from the variable environment at the end of the block's evaluation. In the case of a method invocation there are multiple nested `ret`*s* which pop off the receiver and the method parameters from the variable environment at the end of the method's evaluation. The expression $\texttt{retr}(r, e)$ is used to pop off the top region, r of the store stack at the end of expression e evaluation.

Dynamic semantics rules of region annotated Core-Java are shown in Fig. 7 and Fig. 8. The evaluation judgment is of the form:

$$\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e']$$

where ϖ (ϖ') denotes the store before (after) evaluation, while Π (Π') denotes the variable environment before (after) evaluation. The store ϖ organized as a stack establishes the outlive relations among regions at runtime. The function $ord(\varpi)$ returns the outlive relations for a given store. The function $dom(\varpi)$ returns the set of the store regions, while the function $location\_dom(\varpi)$ returns the set of all locations from the store. They are defined as follows:

$$ord([r_1 \mapsto Rgn_1][r_2 \mapsto Rgn_2]\varpi) =_{def} (r_2 \succeq r_1) \wedge ord([r_2 \mapsto Rgn_2]\varpi)$$
$$ord([r \mapsto Rgn]) =_{def} true \quad ord([\,]) =_{def} true$$
$$dom([r \mapsto Rgn]\varpi) =_{def} \{r\} \cup dom(\varpi) \quad dom([r \mapsto \emptyset]\varpi) =_{def} \{r\} \cup dom(\varpi) \quad dom([\,]) =_{def} \emptyset$$
$$location\_dom(\varpi) =_{def} \{(r, o) \mid \varpi = \varpi_1[r \mapsto Rgn]\varpi_2 \wedge Rgn \neq \emptyset \wedge o \in dom(Rgn)\}$$

Notation $\varpi(r)(o)$ denotes an access into the region r at the offset o, as follows:

$$\varpi(r)(o) =_{def} Rgn(o) \;\; where \;\; \varpi = \varpi_1[r \mapsto Rgn]\varpi_2$$

$$\frac{}{\langle \varpi, \Pi\rangle[v] \hookrightarrow \langle \varpi, \Pi\rangle[\Pi(v)]}\ [\text{D-VAR}]\quad v \in dom(\Pi)$$

$$\frac{\Pi(v)=(r,o) \quad \varpi=\varpi_1[r\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V)}{\langle \varpi, \Pi\rangle[v.f] \hookrightarrow \langle \varpi, \Pi\rangle[V(f)]}\ [\text{D-FD}]$$

$$\frac{\langle \varpi, \Pi\rangle[e] \hookrightarrow \langle \varpi', \Pi'\rangle[e']}{\langle \varpi, \Pi\rangle[lhs = e] \hookrightarrow \langle \varpi', \Pi'\rangle[lhs = e']}\ [\text{D-ASSGN1}]\quad lhs = v \mid v.f$$

$$\frac{v\in dom(\Pi) \quad \Pi'=\Pi+\{v\mapsto\delta\} \quad \delta=(r_1,o_1) \ \wedge\ r_1\in dom(\varpi)}{\langle \varpi, \Pi\rangle[v=\delta] \hookrightarrow \langle \varpi, \Pi'\rangle[()]}\ [\text{D-ASSGN2}]$$

$$\frac{v \in dom(\Pi) \quad \delta=(r_1,o_1) \ \wedge\ r_1\notin dom(\varpi)}{\langle \varpi, \Pi\rangle[v=\delta] \hookrightarrow \texttt{danglingerr}}\ [\text{D-ASSGN2-DANGLERR}]$$

$$\frac{\Pi(v)=(a,o) \quad \varpi=\varpi_1[a\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V) \quad Rgn'=Rgn+\{o\mapsto cn\langle a^+\rangle(V+\{f\mapsto\delta\})\} \quad \varpi'=\varpi_1[a\mapsto Rgn']\varpi_2 \quad \delta=(r_1,o_1) \ \wedge\ \mathbf{ord}(\varpi)\Rightarrow(r_1\succeq\mathbf{fieldregion}(cn\langle a^+\rangle, f))}{\langle \varpi, \Pi\rangle[v.f=\delta] \hookrightarrow \langle \varpi', \Pi\rangle[()]}\ [\text{D-ASSGN3}]$$

$$\frac{\Pi(v)=(a,o) \quad \varpi=\varpi_1[a\mapsto Rgn]\varpi_2 \quad Rgn(o)=cn\langle a^+\rangle(V) \quad \delta=(r_1,o_1) \ \wedge\ \neg\,(\mathbf{ord}(\varpi)\Rightarrow(r_1\succeq\mathbf{fieldregion}(cn\langle a^+\rangle, f)))}{\langle \varpi, \Pi\rangle[v.f=\delta] \hookrightarrow \texttt{danglingerr}}\ [\text{D-ASSGN3-DANGLERR}]$$

$$\frac{\begin{array}{c}\mathbf{class}\ cn\langle r_{1..n}\rangle\ \mathbf{extends}\ c\langle ...\rangle\ \mathbf{where}\ \varphi_{inv}\ \{...\}\in P \qquad \mathbf{ord}(\varpi)\Rightarrow\varphi_{inv} \\ \varpi=\varpi_1[r_1\mapsto Rgn]\varpi_2 \quad V=\{f_1\mapsto\Pi(v_1),...,f_p\mapsto\Pi(v_p)\} \quad \mathbf{fieldlist}(cn\langle r_{1..n}\rangle)=(t_i\,f_i)_{i:1..p} \\ \mathit{if}\ \Pi(v_i)=(r_i',o_i')\ \mathit{then}\ \mathbf{ord}(\varpi)\Rightarrow(r_i'\succeq\mathbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i)) \quad i=1..p \\ o\notin dom(Rgn) \quad Rgn'=Rgn+\{o\mapsto cn\langle r_{1..n}\rangle(V)\} \quad \varpi'=\varpi_1[r_1\mapsto Rgn']\varpi_2\end{array}}{\langle \varpi, \Pi\rangle[\mathbf{new}\ cn\langle r_{1..n}\rangle(v_{1..p})] \hookrightarrow \langle \varpi', \Pi\rangle[(r_1,o)]}\ [\text{D-NEW}]$$

$$\frac{\begin{array}{c}\mathbf{class}\ cn\langle r_{1..n}\rangle\ \mathbf{extends}\ c\langle ...\rangle\ \mathbf{where}\ \varphi_{inv}\ \{...\}\ \in\ P \\ V=\{f_1\mapsto\Pi(v_1),...,f_p\mapsto\Pi(v_p)\} \quad \mathbf{fieldlist}(cn\langle r_{1..n}\rangle)=(t_i\,f_i)_{i:1..p} \\ \neg(\mathbf{ord}(\varpi)\Rightarrow\varphi_{inv})\ \vee\ (\exists i\in\{1..p\}\cdot\Pi(v_i)=(r_i',o_i')\ \wedge \\ \qquad \neg(\mathbf{ord}(\varpi)\Rightarrow(r_i'\succeq\mathbf{fieldregion}(cn\langle r_{1..n}\rangle, f_i)))\end{array}}{\langle \varpi, \Pi\rangle[\mathbf{new}\ cn\langle r_{1..n}\rangle(v_{1..p})] \hookrightarrow \texttt{danglingerr}}\ [\text{D-NEW-DANGLERR}]$$

$$\frac{\begin{array}{c}\{a^+, a'^+\}\subset\mathbf{dom}(\varpi) \\ \Pi(v_0')=(a_1,o) \quad \varpi(a_1)(o)=cn\langle a^+\rangle(V) \\ (t_0\ mn\langle a^+ r'^+\rangle((t\,v)_{1..p})\mathbf{where}\ \varphi\ \{e\})\in cn\langle a^+\rangle \\ n_i=\mathit{fresh}() \ \ i=0..p \quad \rho=[r'^+\mapsto a'^+] \quad \Pi'=\Pi+\{n_i\mapsto\Pi(v_i')_{i:0..p}\} \\ e'=\texttt{ret}(n_0,\texttt{ret}(n_1,..\texttt{ret}(n_p,[\mathit{this}\mapsto n_0][v_i\mapsto n_i]_{i:1}^p\rho e)))\end{array}}{\langle \varpi,\Pi\rangle[v_0'.mn\langle a^+a'^+\rangle(v_{1..p}')] \hookrightarrow \langle \varpi,\Pi'\rangle[e']}\ [\text{D-INVOKE}]$$

$$\frac{\neg(r^+\in\mathbf{dom}(\varpi))}{\langle \varpi,\Pi\rangle[v.mn\langle r^+\rangle(v^*)] \hookrightarrow \texttt{danglingerr}}\ [\text{D-INVOKE-DANGLERR}]$$

Fig. 7. Dynamic Semantics for Region-Annotated Core-Java: Part I

$$\frac{n{=}\mathit{fresh}()\quad \Pi'{=}\Pi{+}\{(n{\mapsto}\mathbf{init}(t))\}\quad e'{=}\mathtt{ret}(n,e)}{\langle\varpi,\Pi\rangle[\{(t\,v)\ e\}]{\hookrightarrow}\langle\varpi,\Pi'\rangle[e']}\ [\textsc{D-EB}] \qquad \frac{\langle\varpi,\Pi\rangle[e]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathtt{ret}(v,e)]{\hookrightarrow}\langle\varpi',\Pi'\rangle[\mathtt{ret}(v,e')]}\ [\textsc{D-RET1}]$$

$$\frac{}{\langle\varpi,\Pi\rangle[\mathtt{ret}(v,\delta)]{\hookrightarrow}\langle\varpi,\Pi-\{v\}\rangle[\delta]}\ [\textsc{D-RET2}]$$

$$\frac{a{=}\mathit{fresh}()}{\langle\varpi,\Pi\rangle[\mathbf{letreg}\ r\ \mathbf{in}\ e]{\hookrightarrow}\langle[a{\mapsto}\emptyset]\varpi,\Pi\rangle[\mathtt{retr}(a,[r{\mapsto}a]e)]}\ [\textsc{D-LETR}]$$

$$\frac{\langle\varpi,\Pi\rangle[e]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e']}{\langle\varpi,\Pi\rangle[\mathtt{retr}(a,e)]{\hookrightarrow}\langle\varpi',\Pi'\rangle[\mathtt{retr}(a,e')]}\ [\textsc{D-RETR1}]$$

$$\frac{\begin{array}{c}(\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))\\ \forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi){\wedge}\\ \forall f\in\mathit{dom}(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\Pi\rangle[\mathtt{retr}(a,\delta)]{\hookrightarrow}\langle\varpi,\Pi\rangle[\delta]}\ [\textsc{D-RETR2}]$$

$$\frac{\begin{array}{c}\neg(a{=}a_1){\vee}\\ \neg((\delta{=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi)))\vee\neg((\forall v{\in}\Pi\cdot(\Pi(v){=}(r,o)){\Rightarrow}(r{\in}\mathbf{dom}(\varpi))))\\ \vee\neg(\forall(r_1,o){\in}\mathit{location\_dom}(\varpi)\cdot(\varpi(r_1)(o){=}cn\langle r_{1..n}\rangle(V)){\Rightarrow}(r_{1..n}{\in}\mathbf{dom}(\varpi){\wedge}\\ \forall f\in\mathit{dom}(V)\ .\ V(f){=}(r_f,o_f)\wedge r_f{\in}\mathbf{dom}(\varpi)))\end{array}}{\langle[a{\mapsto}\mathit{Rgn}]\varpi,\Pi\rangle[\mathtt{retr}(a_1,\delta)]{\hookrightarrow}\mathtt{danglingerr}}\ [\textsc{D-RETR2-DANGLERR}]$$

$$\frac{\Pi(v){=}\mathit{true}}{\langle\varpi,\Pi\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]{\hookrightarrow}\langle\varpi,\Pi\rangle[e_1]}\ [\textsc{D-IF1}] \qquad \frac{\Pi(v){=}\mathit{false}}{\langle\varpi,\Pi\rangle[\mathbf{if}\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]{\hookrightarrow}\langle\varpi,\Pi\rangle[e_2]}\ [\textsc{D-IF2}]$$

$$\frac{\Pi(v){=}\mathit{true}}{\langle\varpi,\Pi\rangle[\mathbf{while}\ v\ e]{\hookrightarrow}\langle\varpi,\Pi\rangle[e\ ;\mathbf{while}\ v\ e]}\ [\textsc{D-LOOP1}] \qquad \frac{\Pi(v){=}\mathit{false}}{\langle\varpi,\Pi\rangle[\mathbf{while}\ v\ e]{\hookrightarrow}\langle\varpi,\Pi\rangle[()]}\ [\textsc{D-LOOP2}]$$

$$\frac{\langle\varpi,\Pi\rangle[e_1]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e_1']}{\langle\varpi,\Pi\rangle[e_1\ ;e_2]{\hookrightarrow}\langle\varpi',\Pi'\rangle[e_1'\ ;e_2]}\ [\textsc{D-SEQ1}] \qquad \frac{}{\langle\varpi,\Pi\rangle[\delta_1\ ;e_2]{\hookrightarrow}\langle\varpi,\Pi\rangle[e_2]}\ [\textsc{D-SEQ2}]$$

$$\frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.f]{\hookrightarrow}\mathtt{nullerr}}\ [\textsc{D-NULLERR1}] \qquad \frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.f=\delta]{\hookrightarrow}\mathtt{nullerr}}\ [\textsc{D-NULLERR2}] \qquad \frac{\Pi(v){=}\mathtt{null}}{\langle\varpi,\Pi\rangle[v.mn\langle a^*\rangle(u^*)]{\hookrightarrow}\mathtt{nullerr}}\ [\textsc{D-NULLERR3}]$$

Fig. 8. Dynamic Semantics for Region-Annotated Core-Java: Part II

We define the meaning of *no-dangling references* property at runtime. The property refers to two kinds of references: (1) references from variable environment to store

locations, and (2) references from store locations to other store locations. Note that the notion of *no-dangling references* was introduced in Fig. 1, and a reference is formalized as a location (r, o).

Definition 1. *(live location) A location* (r, o) *is* live *with respect to a store* ϖ*, if* $r \in dom(\varpi)$.

Definition 2. *(no-dangling)*

1. *A variable environment* Π *is* no-dangling *with respect to a store* ϖ *if for all* $v \in dom(\Pi), \Pi(v)$ *is either a primitive value, or a live location* (r, o) *with respect to* ϖ.
2. *A runtime store* ϖ *is* no-dangling *if each region* $r_1 \in dom(\varpi)$ *contains only references to regions older than itself, that means that for each location* $(r_1, o) \in location\_dom(\varpi)$ *containing an object value* $\varpi(r_1)(o) = cn\langle r_{1..n}\rangle(V)$*, that object value satisfies the* non-dangling requirement for a class*, such that* $ord(\varpi) \Rightarrow \bigwedge_{i:2..n}(r_i \succeq r_1)$ *and the current values of the fields are either primitives or references to regions older than those expected by the region type* $cn\langle r_{1..n}\rangle$*, as follows:*

$$\forall f \in dom(V)\ .\ V(f) = (r_f, o_f) \quad ord(\varpi) \Rightarrow r_f \succeq fieldregion(cn\langle r_{1..n}\rangle, f)$$

Function $fieldregion(cn\langle r_{1..n}\rangle, f)$ *computes the region type of the class field* f *and then returns its first region where the field is expected to be stored.*

The dynamic semantics evaluation rules may yield two possible runtime errors, namely:

$$Error ::= \texttt{nullerr} \mid \texttt{danglingerr}$$

The first error `nullerr` is due to null pointers (by accessing fields or methods of null objects). The second error `danglingerr` is reported when a store updating operation or a variable environment updating operation creates a dangling reference. Our dynamic semantics rules use runtime checks to guarantee that a `danglingerr` error is reported (and the execution is aborted) whenever the program evaluation tries to create a dangling reference. There are five situations that require no-dangling reference checks at runtime:

- *Creation of a new object value.* Rule [D–NEW] checks whether the class invariant holds, $ord(\varpi) \Rightarrow \varphi_{inv}$ (mainly whether the fields regions $r_{i:2..n}$ outlive the region r_1 of the object). The initial value of a field is also checked to be stored in a region that outlives the expected region of that field $r'_i \succeq fieldregion(cn\langle r_{1..n}\rangle, f_i)$. The function $fieldlist(cn\langle r_{1..n}\rangle)$ is defined in Fig. 6.
- *Updating of an object's field.* Rule [D–ASSGN3] checks whether the region r_1 of the new location $\delta = (r_1, o_1)$ outlives the expected region for the object field f, $r_1 \succeq fieldregion(cn\langle a^+\rangle, f)$.
- *Updating a variable from the variable environment.* Rule [D–ASSGN2] checks whether the new location $\delta = (r_1, o_1)$ assigned to a variable is live, namely its region is in the current store, $r_1 \in dom(\varpi)$.
- *Deallocation of a region.* Rule [D–RETR2] checks whether the region a is on the top of the store stack. Then it checks whether a reference to a does not escape neither through the value result δ, nor through the program variable environment Π, nor through the object values of the store ϖ. Note that when a new region is allocated, in rule [D–LETR], a fresh region name is used in order to avoid region name duplication in the store.

– *Calling a method.* Rule [D–INVOKE] checks whether the method's region arguments are in the current store and then prepares the variable environment for the method's body execution.

The corresponding rules [D–NEW–DANGLERR], [D–ASSGN3–DANGLERR], [D–ASSGN2 –DANGLERR], [D–RETR2–DANGLERR], and [D–INVOKE–DANGLERR] generate a `danglingerr` error due to the failure of their runtime checks. In the rules [D–ASSGN2], [D–ASSGN3], and [D–LOOP2] the result () denotes the singleton value of type **void**. Note that the type **void** is assumed to be isomorphic to type **unit**. In rule [D–EB], the locally declared variable is assigned, with the help of the function **init**, an initial value according to its type as follows:

$$\begin{array}{l}\textbf{init}(t) =_{def} \mathit{case}\ t\ \mathit{of}\\ \quad \mathit{boolean} \rightarrow \mathit{false}\\ \quad \mathit{int} \rightarrow 0\\ \quad cn\langle r_{1..n}\rangle \rightarrow \texttt{null}\end{array}$$

5 Extended Static Semantics

In this section we extend our static semantics rules from Section 3 to include the new intermediate constructions introduced by the small-step dynamic semantics rules in Section 4.

First we define a *valid program* using a novel syntactic condition *valid(e)*, that restricts the places where the intermediate constructions may occur in a program.

Definition 3. *(valid program)*

1. *A program is a* valid program *if all the program's classes are valid classes.*
2. *A class is a* valid class *if all the class's methods are valid methods.*
3. *A method is a* valid method *if the method's body e is a valid block expression such that $retvars(e)=\emptyset$ and $retregs(e)=\emptyset$.*
4. *Expression e is a* valid expression *if the predicate $valid(e)$ holds, where $valid(e)$ is defined as follows:*

$$\begin{array}{lll}valid(e) =_{def} \mathit{case}\ e\ \mathit{of} & & \\ \quad \{(t\ v)\ e\} & \rightarrow & retvars(e)=\emptyset \wedge retregs(e)=\emptyset \\ \quad lhs = e & \rightarrow & retvars(e)\cap vars(lhs)=\emptyset \wedge valid(e) \\ \quad e_1\ ;\ e_2 & \rightarrow & retregs(e_2)=\emptyset \wedge retvars(e_2)=\emptyset \wedge valid(e_1) \\ & & \wedge retvars(e_1)\cap vars(e_2)=\emptyset \wedge retregs(e_1)\cap regs(e_2)=\emptyset \\ \quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & retregs(e_1)=\emptyset \wedge retvars(e_1)=\emptyset \\ & & \wedge retregs(e_2)=\emptyset \wedge retvars(e_2)=\emptyset \\ \quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & retregs(e)=\emptyset \wedge retvars(e)=\emptyset \\ \quad \texttt{ret}(v,e) & \rightarrow & v \notin retvars(e) \wedge valid(e) \\ \quad \texttt{retr}(r,e) & \rightarrow & r \notin retregs(e) \wedge valid(e) \\ \quad otherwise & \rightarrow & true\end{array}$$

This condition does not restrict source-level region calculus, since intermediate constructions are generated during the program evaluation. A source language Core-Java program is by default a valid program since it does not contain any intermediate expression. The above condition is based on the functions $vars(e)$, $retvars(e)$, $regs(e)$, and $retregs(e)$ which are defined as follows:

Definition 4. *1. The function* $vars(e)$ *computes the set of all program variables which occur in the expression e, excepting those variables introduced by e's block subexpressions, as follows:*

$$
\begin{array}{lll}
vars(e) =_{def} \textit{case } e \textit{ of} & & \\
\quad \texttt{ret}(v, e) & \rightarrow & \{v\} \cup vars(e) \\
\quad \{(t\ v)\ \ e\} & \rightarrow & vars(e) \setminus \{v\} \\
\quad \texttt{retr}(r, e) \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & vars(e) \\
\quad v.f = e \mid v = e \mid \textbf{while}\ v\ e & \rightarrow & \{v\} \cup vars(e) \\
\quad v.f \mid v & \rightarrow & \{v\} \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & \{v\} \cup vars(e_1) \cup vars(e_2) \\
\quad e_1\ ;\ e_2 & \rightarrow & vars(e_1) \cup vars(e_2) \\
\quad \textbf{new}\ cn\langle r^+\rangle(v^*) & \rightarrow & \{v^*\} \\
\quad v.mn\langle v^*\rangle(v^*) & \rightarrow & \{v\} \cup \{v^*\} \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

2. The function $retvars(e)$ *computes the set of all program variables which occur in the ret subexpressions of the expression e, as follows:*

$$
\begin{array}{lll}
retvars(e) =_{def} \textit{case } e \textit{ of} & & \\
\quad \texttt{ret}(v, e) & \rightarrow & \{v\} \cup retvars(e) \\
\quad \texttt{retr}(r, e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow & retvars(e) \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & retvars(e) \\
\quad e_1\ ;\ e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & retvars(e_1) \cup retvars(e_2) \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

3. The function $regs(e)$ *computes the set of all region variables which occur in the expression e, excepting those regions introduced by e's letreg subexpressions, as follows:*

$$
\begin{array}{lll}
regs(e) =_{def} \textit{case } e \textit{ of} & & \\
\quad \{(t\ v)\ \ e\} & \rightarrow & reg(t) \cup regs(e) \\
\quad \texttt{retr}(r, e) & \rightarrow & \{r\} \cup regs(e) \\
\quad \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & regs(e) \setminus \{r\} \\
\quad \texttt{ret}(v, e) \mid v.f = e \mid v = e \mid \textbf{while}\ v\ e & \rightarrow & regs(e) \\
\quad (r, o) & \rightarrow & \{r\} \\
\quad \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \mid e_1\ ;\ e_2 & \rightarrow & regs(e_1) \cup regs(e_2) \\
\quad \textbf{new}\ cn\langle r^+\rangle(v^*) \mid v.mn\langle r^+\rangle(v^*) & \rightarrow & \{r^+\} \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

where reg(t) is defined in the Figure 6.

4. The function $retregs(e)$ *computes the set of all region variables which occur in the retr subexpressions of the expression e, as follows:*

$$
\begin{array}{lll}
retregs(e) =_{def} \textit{case } e \textit{ of} & & \\
\quad \texttt{retr}(r, e) & \rightarrow & \{r\} \cup retregs(e) \\
\quad \texttt{ret}(v, e) \mid v.f = e \mid v = e \mid \{(t\ v)\ e\} & \rightarrow & retregs(e) \\
\quad \textbf{while}\ v\ e \mid \textbf{letreg}\ r\ \textbf{in}\ e & \rightarrow & retregs(e) \\
\quad e_1\ ;\ e_2 \mid \textbf{if}\ v\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \rightarrow & retregs(e_1) \cup retregs(e_2) \\
\quad otherwise & \rightarrow & \emptyset
\end{array}
$$

In order to describe the type of each location, we introduce a *store typing*. This ensures that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. Store typing is conventionally used to link static and dynamic semantics [20]. In our case, it is denoted by Σ, as follows:

$$\Sigma \in \mathit{StoreType} = \mathit{RegVar} \rightharpoonup_{\text{fin}} \mathit{Offset} \rightharpoonup_{\text{fin}} \mathit{Type}$$

The judgments of static semantics are extended with store typing, as follows:

$$P; \Gamma; R; \varphi; \Sigma \vdash e : t$$

For a store typing $\Sigma : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type}$, a region r, a location (r, o), and a type t we also introduce the following notations:

$$\begin{array}{l}
\mathit{dom}(\Sigma) = R \qquad \Sigma(r)(o) = f(o),\ \mathit{where}\ f = \Sigma(r) \\
\mathit{location_dom}(\Sigma) =_{\mathit{def}} \{(r, o) \mid r \in \mathit{dom}(\Sigma) \wedge f = \Sigma(r) \wedge f \neq \emptyset \wedge o \in \mathit{dom}(f)\} \\
\Sigma - r =_{\mathit{def}} \Sigma_1\ \mathit{such\ that}\ \Sigma_1 : (R - \{r\}) \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \wedge \forall r' \in (R - r) \cdot \Sigma_1(r') = \Sigma(r') \\
\Sigma + r =_{\mathit{def}} \Sigma_2\ \mathit{such\ that}\ \Sigma_2 : (R \cup \{r\}) \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \wedge \Sigma_2(r) = \emptyset \wedge \forall r' \in R \cdot \Sigma_2(r') = \Sigma(r') \\
\Sigma - (r, o) =_{\mathit{def}} \Sigma_3\ \mathit{such\ that}\ \Sigma_3 : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge r \in R \wedge \Sigma_3(r) = \Sigma(r) - \{o\} \wedge \forall r' \in (R - r) \cdot \Sigma_3(r') = \Sigma(r') \\
\Sigma + ((r, o) : t) =_{\mathit{def}} \Sigma_4\ \mathit{such\ that}\ \Sigma_4 : R \rightharpoonup_{\text{fin}} O \rightharpoonup_{\text{fin}} \mathit{Type} \\
\qquad \wedge r \in R \wedge \Sigma_4(r) = \Sigma(r) + \{o \mapsto t\} \wedge \forall r' \in (R - r) \cdot \Sigma_4(r') = \Sigma(r')
\end{array}$$

The judgments of the new intermediate expressions are presented in Fig. 9. They assume that the expressions are valid with respect to the Definition 3. The first two rules [RC–LOCATION] and [RC–ObjVal] are used to type the store, either a location or an object value (i.e. a location's content). Rule [RC–ObjVal] preserves the same invariants as those of the rule [RC–NEW]. Rule [RC–RET] ensures that the variable to be popped off, v is in the current environment Γ. The subsumption rule [SUBSUMPTION] simplifies the next theorems and their proofs.

Rule [RC–RETR] is similar to rule [RC–LETR], but it takes into account the evaluation of the expression $\texttt{retr}(r, e)$. The first check ensures that the region to be deallocated,

$$\frac{r \in R \quad \Sigma(r)(o) = t}{P; \Gamma; R; \varphi; \Sigma \vdash (r, o) : t}\ [\text{RC–LOCATION}]$$

$$\frac{P; R; \varphi \vdash_{\mathit{type}} cn\langle r_{1..n}\rangle \quad \mathit{fieldlist}(cn\langle r_{1..n}\rangle) = (t_i\, f_i)_{i:1..p} \quad P; \Gamma; R; \varphi; \Sigma \vdash V(f_i) : t'_i \quad P; R; \varphi \vdash t'_i <: t_i \quad i{=}1..p}{P; \Gamma; R; \varphi; \Sigma \vdash cn\langle r_{1..n}\rangle(V) : cn\langle r_{1..n}\rangle}\ [\text{RC–ObjVal}]$$

$$\frac{v \in \Gamma \quad P; \Gamma; R; \varphi; \Sigma \vdash e : t}{P; \Gamma; R; \varphi; \Sigma \vdash \texttt{ret}(v, e) : t}\ [\text{RC–RET}]$$

$$\frac{P; \Gamma; R; \varphi; \Sigma \vdash e : t' \quad P; R; \varphi \vdash t' <: t}{P; \Gamma; R; \varphi; \Sigma \vdash e : t}\ [\text{SUBSUMPTION}]$$

$$\frac{\begin{array}{c} a \in R \quad R_t = R - \mathit{lreg}(e) - \{a\} \quad \varphi \Rightarrow \bigwedge_{r \in R_t} (r \succeq a) \\ \mathit{reg}(t) \subseteq R_t \quad \mathit{reg}(\Gamma - \mathit{lvar}(e)) \subseteq R_t \quad P; \Gamma; R; \varphi; \Sigma \vdash e : t \end{array}}{P; \Gamma; R; \varphi; \Sigma \vdash \texttt{retr}(a, e) : t}\ [\text{RC–RETR}]$$

Fig. 9. Region Type Checking Rules for Valid Intermediate Expressions

a is in R. The R_t denotes the regions from R which are different than a and are not younger than a. Note that $lreg(e)$ denotes the regions which are younger than a. The second check ensures that our type system uses only lexically scoped regions such that the region to be deallocated, a is always on the top of the regions stack. The third and the fourth check ensure that the region a and the regions younger than a do not escape either through the result or through the live variables of the type environment. Note that $lvar(e)$ denotes the local variables of the expression e which are deallocated from the variable environment during the evaluation of e.

The rules from Fig. 9 are using the functions $lvar(e)$, $lreg(e)$, and $lloc(e)$ which are defined as follows:

Definition 5. *Using the evaluation rules from Fig. 7 and Fig. 8*

1. *The function $lvar(e)$ estimates the set of variables which may be popped off from the variable environment Π during the evaluation of the valid expression e (note that only $\texttt{ret}(v, e)$ may affect Π), as follows:*

$$\begin{array}{lll} lvar(e) =_{def} \textit{case } e \textit{ of} & & \\ \quad \texttt{ret}(v, e) & \rightarrow & \{v\} \cup lvar(e) \\ \quad \texttt{retr}(r, e) \mid lhs = e \mid e\,;e_1 & \rightarrow & lvar(e) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

2. *The function $lreg(e)$ estimates the set of regions which may be popped off from the store ϖ during the evaluation of the valid expression e (note that only $\texttt{retr}(r, e)$ may affect ϖ), as follows:*

$$\begin{array}{lll} lreg(e) =_{def} \textit{case } e \textit{ of} & & \\ \quad \texttt{retr}(r, e) & \rightarrow & \{r\} \cup lreg(e) \\ \quad \texttt{ret}(v, e) \mid lhs = e \mid e\,;e_1 & \rightarrow & lreg(e) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

3. *The function $lloc(e)$ estimates the new location which may be created into an existing region during one evaluation step of the valid expression e (note that only* **new** *may create a new location), as follows:*

$$\begin{array}{lll} lloc(e) =_{def} \textit{case } e \textit{ of} & & \\ \quad \textbf{new}\ cn\langle r_1, .., r_n\rangle(v^*) & \rightarrow & \{(r_1, o)\} \\ \quad \texttt{ret}(v, e) \mid \texttt{retr}(r, e) \mid lhs = e \mid e\,;e_1 & \rightarrow & lloc(e) \\ \quad \textit{otherwise} & \rightarrow & \emptyset \end{array}$$

where the offset o of the region r is the offset where the next allocation in r is done.

6 Soundness Theorems

In this section we prove the soundness of our region calculus, namely that a valid program well-typed by our type system never creates dangling references. We use a syntactic proof method [25], based on a subject reduction theorem and a progress theorem.

First we define the consistency relationship between the static and dynamic semantics, namely a relationship between what we can estimate at compile-time and what can happen during run-time execution.

Definition 6. *(consistency relationship)*
A run-time environment (ϖ, Π) *is* consistent *with a static environment* $(\Gamma, R, \varphi, \Sigma)$*, written* $\Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle$*, if the following judgment holds:*

$$dom(\Gamma)=dom(\Pi) \qquad \forall v \in dom(\Pi) \cdot P; \Gamma; R; \varphi; \Sigma \vdash \Pi(v) : \Gamma(v) \qquad reg(\Gamma) \subseteq R$$
$$location\_dom(\Sigma)=location\_dom(\varpi) \qquad dom(\Sigma)=dom(\varpi) \qquad R=dom(\varpi)$$
$$ord(\varpi) \Rightarrow \varphi \qquad \forall (r,o) \in location\_dom(\varpi) \cdot P; \Gamma; R; \varphi; \Sigma \vdash \varpi(r)(o) : \Sigma(r)(o)$$

Note that $\varpi(r)(o)$ returns an object value $cn\langle r^* \rangle(V)$ whose type is $cn\langle r^* \rangle$. In our instrumented operational semantics an object value and its type are stored together.

The subject reduction theorem ensures that the region type is preserved during the execution of a valid program, as follows:

Theorem 1. *(Subject Reduction): If*

$$valid(e) \quad P; \Gamma; R; \varphi; \Sigma \vdash e : t$$
$$\Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle$$
$$\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$$

then there exist Σ'*,* Γ'*,* R'*, and* φ'*, such that*

$$(\Sigma' - (lreg(e') - lreg(e))) - (lloc(e) - lloc(e')) = \Sigma - (lreg(e) - lreg(e'))$$
$$\Gamma' - (lvar(e') - lvar(e)) = \Gamma - (lvar(e) - lvar(e'))$$
$$R' - (lreg(e') - lreg(e)) = R - (lreg(e) - lreg(e'))$$
$$\varphi' - (lreg(e') - lreg(e)) \Rightarrow \varphi - (lreg(e) - lreg(e'))$$
$$\Gamma', R', \varphi', \Sigma' \vDash \langle \varpi', \Pi' \rangle$$
$$valid(e') \quad P; \Gamma'; R'; \varphi'; \Sigma' \vdash e' : t.$$

Proof: By structural induction on e. The detailed proof is in [11].

Although the hypothesis of the above theorem contains an evaluation relation, the proof does not use the run-time checks associated with the evaluation rules to prove that the result of the evaluation (result and dynamic environment) is well-typed, valid and consistent.

The progress theorem guarantees that the execution of a valid program cannot generate `danglingerr` errors, by proving that those run-time checks are redundant for a well-typed valid program (the run-time checks are proved by the static semantics).

Theorem 2. *(Progress) If*

$$valid(e) \quad P; \Gamma; R; \varphi; \Sigma \vdash e : t$$
$$\Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle$$

then either

- *e is a value, or*
- $\langle \varpi, \Pi \rangle[e] \hookrightarrow$ `nullerr` *or*
- *there exist* ϖ'*,* Π'*,* e' *such that* $\langle \varpi, \Pi \rangle[e] \hookrightarrow \langle \varpi', \Pi' \rangle[e']$.

Proof: By induction over the depth of the type derivation for expression e. The detailed proof is in [11].

We conclude with the following soundness theorem for region annotated Core-Java. The theorem states that if a valid program is well-typed and is evaluated in a run-time

environment consistent with the static environment, the result of a finite number of reduction steps (denoted by $\hookrightarrow^*$) is (1) either an error different from a dangling error, (2) or a value, (3) or that the program diverges (namely after a finite number of reduction steps there still exists one more reduction step). The evaluation never reports dangling errors, namely the program never creates dangling references.

Theorem 3. *(Soundness) Given a well-typed valid Core-Java program P=def* and the main function (void* **main**(*void*)$\{e_0\}$)$\in P$, where e_0 is a well-typed valid closed term (without free regions and free variables), such that retvars$(e_0)=\emptyset \wedge$ retregs$(e_0)=\emptyset$ and $P; \Gamma_0;\ R_0; \varphi_0; \Sigma_0 \vdash e_0 : void$, where $\Gamma_0=\emptyset$, $R_0=\emptyset$, $\varphi_0=true$, and $\Sigma_0=\emptyset$. Starting from the initial run-time environment $\langle \varpi_0, \Pi_0 \rangle$, where $\varpi_0=[\,]$, $\Pi_0=\emptyset$, such that $\Gamma_0, R_0, \varphi_0, \Sigma_0 \vDash \langle \varpi_0, \Pi_0 \rangle$. Then either*

$$\langle \varpi_0, \Pi_0 \rangle [e_0] \hookrightarrow^* \texttt{nullerr} \tag{1}$$

or there exist a store ϖ, a variable environment Π, a value δ, a type environment Γ, a set of regions R, a region constraint φ, a store typing Σ such that

$$\langle \varpi_0, \Pi_0 \rangle [e_0] \hookrightarrow^* \langle \varpi, \Pi \rangle [\delta] \quad \Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle \quad P; \Gamma; R; \varphi; \Sigma \vdash \delta : void \tag{2}$$

or for a store ϖ, a variable environment Π, a valid expression e, a type environment Γ, a set of regions R, a region constraint φ, a store typing Σ such that

$$\langle \varpi_0, \Pi_0 \rangle [e_0] \hookrightarrow^* \langle \varpi, \Pi \rangle [e] \quad \Gamma, R, \varphi, \Sigma \vDash \langle \varpi, \Pi \rangle \quad P; \Gamma; R; \varphi; \Sigma \vdash e : void \quad valid(e)$$

there exist a store ϖ', a variable environment Π', an expression e', a type environment Γ', a set of regions R', a region constraint φ', a store typing Σ' such that

$$\langle \varpi, \Pi \rangle [e] \hookrightarrow \langle \varpi', \Pi' \rangle [e'] \quad \Gamma', R', \varphi', \Sigma' \vDash \langle \varpi', \Pi' \rangle \quad P; \Gamma'; R'; \varphi'; \Sigma' \vdash e' : void \quad valid(e') \tag{3}$$

Proof: The proof is an induction on the number of the reduction steps. We can repeatedly use the progress theorem (Theorem 2) to prove that there is a reduction step and then the preservation theorem (Theorem 1) to prove that the run-time environment after evaluation is still well-typed and the evaluation result is valid.

7 Conclusion

We have considered a region calculus consisting of an object-oriented core language annotated with regions. We have defined the dynamic semantics for our region calculus based on a simpler small-step rewriting relation. Some of the region calculus constructions (e.g. `letreg`) are firstly evaluated to intermediate constructions. Therefore the static semantics must also be extended to include these new intermediate constructions. We have used a novel syntactic condition (*valid(e)*) to restrict the places where the intermediate constructions may occur in a program. This condition does not restrict source-level region calculus, since intermediate constructions are generated during the program evaluation. Our dynamic semantics is instrumented with runtime checks to guarantee that a special `danglingerr` error is reported whenever the program evaluation tries to create a dangling reference. We have defined an important consistency relationship between the static and dynamic semantics. A store typing technique is used

to ensure that objects created in the store during run-time are type-wise consistent with those captured by the static semantics. We have proven the soundness of the region calculus by using a syntactic proof method [25], based on subject reduction and progress. The subject reduction theorem ensures that the region type of a valid program is preserved during the evaluation. The progress theorem guarantees that the evaluation of a valid program cannot generate `danglingerr` errors (namely those runtime checks are redundant for a well-typed valid program). We have proven both theorems in a modular fashion using just a simple induction. This simple soundness proof adds confidence to our region-based memory inference and execution systems.

References

1. Bacon, D.F., Cheng, P., Rajan, V.T.: A real-time garbage collector with low overhead and consistent utilization. In: POPL, pp. 285–298 (2003)
2. Birkedal, L., Tofte, M.: A constraint-based region inference algorithm. Theoretical Computer Science 258(1–2), 299–392 (2001)
3. Bollella, G., Brosgol, B., Dibble, P., Furr, S., Gosling, J., Hardin, D., Turnbull, M.: The Real-Time Specification for Java. Addison-Wesley, Reading (2000)
4. Boudol, G.: Typing safe deallocation. In: European Symposium on Programming (ESOP), pp. 116–130 (2008)
5. Boyapati, C., Salcianu, A., Beebee, W., Rinard, M.: Ownership Types for Safe Region-Based Memory Management in Real-Time Java. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 324–337 (2003)
6. Calcagno, C.: Stratified operational semantics for safety and correctness of the region calculus. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 155–165 (2001)
7. Calcagno, C., Helsen, S., Thiemann, P.: Syntactic type soundness results for the region calculus. Information and Computation 173(2), 199–221 (2002)
8. Chin, W.-N., Craciun, F., Qin, S., Rinard, M.C.: Region inference for an object-oriented language. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 243–254 (2004)
9. Christiansen, M.V., Velschow, P.: Region-Based Memory Management in Java. Master's Thesis, Department of Computer Science (DIKU), University of Copenhagen (1998)
10. Craciun, F., Goh, H.Y., Chin, W.-N.: A framework for object-oriented program analyses via Core-Java. In: IEEE International Conference on Intelligent Computer Communication and Processing (ICCP), Cluj-Napoca, Romania, pp. 197–205 (2006)
11. Craciun, F., Qin, S., Chin, W.-N.: A Formal Soundness Proof of Region-based Memory Management for Object-Oriented Paradigm. Technical report, Department of Computer Science, Durham University, UK (April 2008), `http://www.durham.ac.uk/shengchao.qin/papers/reg_cal_proof.pdf`
12. Elsman, M.: Garbage collection safety for region-based memory management. In: ACM Workshop on Types in Language Design and Implementation (TLDI), pp. 123–134 (2003)
13. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-Based Memory Management in Cyclone. In: ACM Conference on Programming Language Design and Implementation (PLDI), pp. 282–293 (2002)
14. Helsen, S.: Region-Based Program Specialization. PhD thesis, Universität Freiburg (2002)
15. Helsen, S., Thiemann, P.: Syntactic type soundness for the region calculus. Electronic Notes in Theoretical Computer Science 41(3) (2000)

16. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp. 132–146 (1999)
17. Morrisett, G.: Compiling with Types. PhD thesis, Carnegie Mellon University (1995)
18. Morrisett, J.G., Felleisen, M., Harper, R.: Abstract Models of Memory Management. In: ACM Conference Conference on Functional Programming Languages and Computer Architecture (FPCA), pp. 66–77 (1995)
19. Niss, H.: Regions are imperative. Unscoped regions and control-sensitive memory management. PhD thesis, University of Copenhagen (2002)
20. Pierce, B.: Types and Programming Languages. MIT Press, Cambridge (2002)
21. Tofte, M., Birkedal, L.: A region inference algorithm. ACM Transactions on Programming Languages and Systems (TOPLAS) 20(4), 734–767 (1998)
22. Tofte, M., Talpin, J.: Implementing the Call-By-Value λ-calculus Using a Stack of Regions. In: ACM Symposium on Principles of Programming Languages (POPL), pp. 188–201 (1994)
23. Tofte, M., Talpin, J.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
24. Wilson, P.R.: Uniprocessor garbage collection techniques. In: International Workshop on Memory Management (IWMM), pp. 1–42 (1992)
25. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information Computation 115(1), 38–94 (1994)

Program Models for Compositional Verification*

Marieke Huisman[1], Irem Aktug[2], and Dilian Gurov[2]

[1] INRIA Sophia Antipolis, France
[2] Royal Institute of Technology, Stockholm, Sweden

Abstract. Compositional verification is crucial for guaranteeing the security of systems where new components can be loaded dynamically. In earlier work, we developed a compositional verification principle for control-flow properties of sequential control flow graphs with procedures. This paper discusses how the principle can be generalised to richer program models. We first present a generic program model, of which the original program model is an instantiation, and explicate under what conditions the compositional verification principle applies. We then present two other example instantiations of the generic model: with exceptional and with multi-threaded control flow, and show that for these particular instantiations the conditions hold. The program models we present are specifically tailored to our compositional verification principle; however, they are sufficiently intuitive and standard to be useful on their own. Tool support and practical application of the method are discussed.

1 Introduction

Compositional verification addresses the problem of proving the correctness of a compound system based on properties of its components. Compositional verification techniques allow one to guarantee that if the new applications satisfy certain local requirements, the global security (policy) of the system is not violated. Such techniques are crucial to ensure the security of any platform, where new applications can be installed dynamically. Typical application areas are *e.g.*, mobile computing, and dynamically reconfiguring distributed systems.

We are interested in both structural and behavioural control flow properties of programs. A structural property is a property of the (finite) flow graph itself, such as "every path from the entry of method m_1 to a call instruction to method m_2 passes a call instruction to method m_3". A behavioural property is a property of the (infinite state) behaviour induced by the flow graph, such as "in any execution of the program, method m_1 calls method m_2 at most once".

In earlier work, we developed a compositional verification method for programs with procedures. Our method supports the following abstract compositional verification principle, where $\mathcal{G}_1$ and $\mathcal{G}_2$ are programs with procedures

* This work was funded in part by the IST programme of the EC, under the IST-FET-2005-015905 MOBIUS project and under the IST-STREP-27004 S3MS project.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 147–166, 2008.

(*i.e.*, components), modelled as *control flow graphs*, and $\uplus$ denotes flow graph composition:

$$\frac{\mathcal{G}_1 \models \sigma \qquad \theta_{I_{\mathcal{G}_1}}(\sigma) \uplus \mathcal{G}_2 \models \phi}{\mathcal{G}_1 \uplus \mathcal{G}_2 \models \phi} \tag{1}$$

Informally, this rule says that to prove that the composition $\mathcal{G}_1 \uplus \mathcal{G}_2$ satisfies property ϕ, it is sufficient to find a "local" property σ of flow graph $\mathcal{G}_1$ (typically a still unavailable component) for which one can verify that: *(i)* σ indeed holds for $\mathcal{G}_1$, and *(ii)* the local property ensures the global property. Task *(i)* is deferred until component $\mathcal{G}_1$ becomes available. Task *(ii)* assumes knowledge of the names of the provided and required methods of $\mathcal{G}_1$ (its so-called *flow graph interface* $I_{\mathcal{G}_1}$), and is achieved by constructing a maximal flow graph for the local property, *i.e.*, $\theta_{I_{\mathcal{G}_1}}(\sigma)$ and by showing that its composition with $\mathcal{G}_2$ satisfies ϕ. In both tasks, the verifications can be performed algorithmically, using finite-state and pushdown automata-based model checking, respectively.

A maximal flow graph *w.r.t.* a property σ is a flow graph that simulates all other flow graphs satisfying property σ. This notion is based on the notion of *maximal model* [9], but in addition takes the set of provided and required methods, *i.e.,* the flow graph interface, into account: a maximal flow graph only simulates flow graphs with the same interface. Our technique requires the local requirement σ to be a structural property, while the global requirement ϕ can be either a structural or a behavioural property. This has the advantage that the approach works for relatively simple program models. All formulae are expressed in the fragment of the modal μ-calculus [14] with boxes and greatest fixed-points only. Recently, we have developed a translation from behavioural properties into structural ones [10]. This allows to apply the compositional verification principle also for local behavioural properties, and thus to (indirectly) reuse the global guarantee as a local assumption for the verification of a larger system. However, in this paper, we do not further discuss this, and we simply assume all local properties to be structural.

We have shown soundness and completeness of the compositional verification principle for a basic program model, only considering sequential control flow. This paper discusses under what conditions the principle can be used with finer, more complex program models. For this, we first present a generic program model, and then explicate the conditions that have to be satisfied by each concrete instantiation. To illustrate the approach, we present two concrete instantiations, one extending the basic program model with exceptional control flow and one with multi-threaded control flow. These finer program models are especially tailored to satisfy the above-mentioned conditions, but are intuitive enough to be useful on their own. In addition, we illustrate how extending the program model allows to express (and verify) more complex program properties.

To support our compositional verification method we have developed a tool set. Originally, this was tailored to the basic program model. The basic version of the tool set has been used to demonstrate utility of the method on an industrial smart card case study [11]. This paper contains an overview of the tool set and describes how various parts of it are adapted to support the new instantiations.

Related Work. The maximal model technique for compositional verification is originally developed by Grumberg and Long [9] for the universal fragment of CTL, and later generalised by Kupferman and Vardi [15] for ACTL*. We have adapted the technique to the fragment of the modal μ-calculus with boxes and greatest fixed-points [11]. Our original program model has been inspired by the one of Besson *et al.* [2], who address the problem of verifying stack invariants of Java programs. The model of Recursive State Machines, proposed by Alur *et al.* [1] is also close to ours, while somewhat finer. However, the authors do not address compositional verification of programs with recursion. Still other models exist for capturing the control flow of applications in Java-like languages, see *e.g.,* [18]. However, because of the specific requirements of our compositional verification technique, we cannot directly reuse these models, and instead rely on our own. Several tools exist for the (non-compositional) verification of behavioural program properties. For example, Moped [13,7] and Alfred [20] encode the behaviour of a program as a pushdown system, that is model checked. In particular, the jMoped variation [23] translates Java bytecode to a pushdown system extended with a set of variables, where instructions are directly mapped to transitions of the system. Also closely related is the two-step extraction technique of Obdržálek [19], where a control flow graph of the program is produced first, and the pushdown system is then generated from this graph. However, neither of these translations addresses multi-threading. Further, existing model checkers for multi-threaded Java (such as Bogor[1] and JavaPathFinder[2]) typically use an implicit program representation that is close to the program itself. Then, abstraction is applied to make verification feasible. In contrast, our program model directly abstracts the program behaviour; without this abstraction a maximal flow graph cannot be constructed.

Overview of the paper. Section 2 describes the generic framework for compositional verification, and shows how our original program model is an instance of this. Sections 3 and 4 describe instantiations with exceptional control flow, and with multi-threaded control flow. Finally, Section 5 draws conclusions and discusses other possible instantiations.

2 A Framework for Compositional Verification

This section presents a method for compositional verification of control flow properties based on a generic program model, identifies sufficient conditions for soundness and completeness of the method, instantiates the generic model to the basic model used in [11], and also outlines the tool set supporting this method.

2.1 Program Model

As the basis for our program model, we use a general notion of specification. Both control flow graph structure and behaviour are defined in terms of such specifications. For a detailed account of the basic definitions, we refer to [11].

[1] See `http://bogor.projects.cis.ksu.edu`
[2] See `http://javapathfinder.sourceforge.net`

Definition 1 (Specification). *A* model *over a set of labels L and a set of atomic propositions A is a structure $\mathcal{M} = (S, L, \rightarrow, A, \lambda)$, where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a labelled transition relation, and $\lambda\colon S \rightarrow \mathcal{P}(A)$ a valuation assigning to each state a set of atomic propositions. A* specification *$\mathcal{S}$ is a pair $(\mathcal{M}, \mathbb{E})$, with $\mathcal{M}$ a model and $\mathbb{E} \subseteq S$ a set of* entry *states.*

The *reachable part* of a specification $\mathcal{S} = (\mathcal{M}, \mathbb{E})$ is defined by $\mathcal{R}(\mathcal{S}) = (\mathcal{M}', \mathbb{E})$, where $\mathcal{M}'$ is obtained from $\mathcal{M}$ by deleting all states and transitions not reachable from $\mathbb{E}$. The *disjoint union* of two specifications is defined by $(\mathcal{M}_1, \mathbb{E}_1) \uplus (\mathcal{M}_2, \mathbb{E}_2) = (\mathcal{M}_1 \uplus \mathcal{M}_2, \mathbb{E}_1 \uplus \mathbb{E}_2)$, where $\mathcal{M}_1 \uplus \mathcal{M}_2 = (S_1 \uplus S_2, L_1 \cup L_2, \{in_i(s) \xrightarrow{a} in_i(s') | s \xrightarrow{a} s' \in \mathcal{M}_i\}, A_1 \cup A_2, \lambda)$, where $\lambda(in_i(s)) = \lambda_i(s)$ and in_i (for $i \in \{1, 2\}$) injects S_i into $S_1 \uplus S_2$. The definition of *simulation* between specifications is standard. Notice that simulation is preserved by disjoint union.

$$\mathcal{S}_1 \leq \mathcal{T}_1 \wedge \mathcal{S}_2 \leq \mathcal{T}_2 \Rightarrow \mathcal{S}_1 \uplus \mathcal{S}_2 \leq \mathcal{T}_1 \uplus \mathcal{T}_2 \tag{2}$$

Let $\mathcal{M}eth$ be an infinite set of method names, and let $\mathcal{C}ontr$ be a possibly infinite set of control values (disjoint from $\mathcal{M}eth$) specific for each instantiation of the model (in the program model with exceptions, for instance, it is a set of exception names). Both sets should be disjoint from any reserved symbols. Every control flow graph comes equipped with an interface, specifying the provided and required methods, and the set of legal control values.

Definition 2 (Flow Graph Interface). *A* flow graph interface *is a triple $I = (I^+, I^-, C)$, where $I^+, I^- \subseteq \mathcal{M}eth$ are finite sets of names of* provided *and* required *methods, and $C \subseteq \mathcal{C}ontr$ is a finite set of control values, respectively. We say I is* closed *if $I^- \subseteq I^+$. The* composition *of two interfaces $I_1 = (I_1^+, I_1^-, C_1)$ and $I_2 = (I_2^+, I_2^-, C_2)$ is defined by $I_1 \cup I_2 = (I_1^+ \cup I_2^+, I_1^- \cup I_2^-, C_1 \cup C_2)$.*

The definition of control flow graph structure, or *flow graphs* for short, is also relativised on the notion of *method specification*, which is specific for each concrete instantiation of the generic program model. We require a method specification to be defined as an instance of the general notion of specification. Given such a definition, one can formally define the notion of *flow graph with interface*.

Definition 3 (Flow Graph). Flow graphs $\mathcal{G}$ with interface I, *written $\mathcal{G} : I$, are inductively defined by*

- *$(\mathcal{M}_m, \mathbb{E}_m) : (\{m\}, M, C)$ if $(\mathcal{M}_m, \mathbb{E}_m)$ is a method specification for m over M and C,*
- *$\mathcal{G}_1 \uplus \mathcal{G}_2 : I_1 \cup I_2$ if $\mathcal{G}_1 : I_1$ and $\mathcal{G}_2 : I_2$.*

A flow graph $\mathcal{G} : I$ is *closed* if its interface I is closed. We use $\leq_s$ to denote *structural simulation* between flow graphs.

Basic Program Model. The compositional verification principle is originally defined for an instance of the generic definition of flow graph, with $\mathcal{C}ontr$ the empty set. In this basic program model, method flow graphs are defined as follows.

```
class Number {

   public static boolean even(int n){
      if (n == 0)
         return true;
      else
         return odd(n-1);
   }

   public static boolean odd(int n){
      if (n == 0)
         return false;
      else
         return even(n-1);

   }
}
```

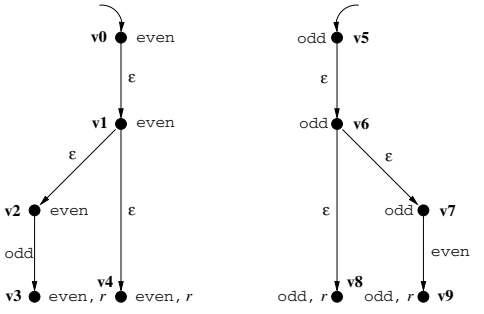


Fig. 1. A simple Java class and its flow graph

Definition 4. (Method Specification) *A* flow graph *for* $m \in \mathcal{M}eth$ *over a set* $M \subseteq \mathcal{M}eth$ *is a finite model* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$*, with* V_m *the set of control nodes of* m*,* $L_m = M \cup \{\varepsilon\}$*,* $A_m = \{m, r\}$*, and* $\lambda_m : V_m \rightarrow \mathcal{P}(A_m)$*, so that* $m \in \lambda_m(v)$ *for all* $v \in V_m$ (i.e., *every node is tagged with its method name). The nodes* $v \in V_m$ *with* $r \in \lambda_m(v)$ *are* return points. *A* method specification *for* $m \in \mathcal{M}eth$ *over* M *is a pair* $(\mathcal{M}_m, \mathbb{E}_m)$ *s.t.* $\mathcal{M}_m$ *is a flow graph for* m *over* M *and* $\mathbb{E}_m \subseteq V_m$ *a non-empty set of* entry points *of* m*.*

Thus, in this program model, a flow graph $\mathcal{G} : I$ is a model over $I^- \cup \{\varepsilon\}$ and $I^+ \cup \{r\}$.

Example 1. Figure 1 shows a simple Java class and the (simplified) flow graph it induces in the basic program model. The flow graph consists of two method specifications - one for method `even` and one for method `odd`. Entry nodes are depicted as usual through edges without source.

2.2 Model Extraction

The tool set that we developed to support our compositional verification technique contains the *Program Analyser (PA),* that extracts flow graphs from Java (bytecode) classes. One can always extract a flow graph that over-approximates the actual behaviour as specified by the Java semantics; the precision of the over-approximation depends on the precision of the static analysis used by PA. PA is built on top of the Soot Java Optimization Framework [24]. Soot transforms a bytecode program into Jimple basic blocks. Then, it makes a class hierarchy analysis, producing a safe over-approximation of the application's call graph. For example, if the analysis cannot determine the receiver of a virtual method call, a call edge is generated for every possible method implementation. Further, Soot produces a control flow graph for each method, abstracting away all values. PA transforms these, using information from the call graph, into flow graphs in the format of the program model. Extending PA to the different instantiations amounts to using additional information produced by Soot's different analyses when translating control flow graphs into flow graphs for the program model.

2.3 Flow Graph Behaviour

Next, we define the behaviour of flow graphs. Since the local guarantees must be properties over the flow graph structure, we only have to define the behaviour of closed flow graphs. The behaviour of a flow graph $\mathcal{G}$, denoted $b(\mathcal{G})$, should also be defined as an instance of the general notion of specification. Also on the behavioural level, we instantiate the definition of simulation $\leq_b$: $\mathcal{G}_1 \leq_b \mathcal{G}_2 \Leftrightarrow b(\mathcal{G}_1) \leq b(\mathcal{G}_2)$. For the compositional verification principle to apply for a concrete program model, structural simulation should imply behavioural simulation:

$$\mathcal{G}_1 \leq_s \mathcal{G}_2 \Rightarrow \mathcal{G}_1 \leq_b \mathcal{G}_2 \tag{3}$$

Basic Program Model The behaviour of the basic flow graphs (where $\mathcal{C}ontr = \varnothing$) is defined as follows.

Definition 5. (Behaviour) *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : (I^+, I^-)$ be a closed flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{bs}, A_b, \lambda_b)$, s.t. $S_b = V \times V^*$, that is, states are* configurations *of control points and stacks, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$, and $\rightarrow_{bs}$ is defined as follows:*

$$\begin{array}{lll}
\text{[transfer]} & (v,\sigma) \xrightarrow{\tau}_{bs} (v',\sigma) & \text{if } v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \\
\text{[call]} & (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_{bs} (v_2, v_1' \cdot \sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \xrightarrow{m_2}_{m_1} v_1',\ v_1 \models \neg r, \\
 & & \quad v_2 \models m_2,\ v_2 \in E \\
\text{[return]} & (v_2, v_1 \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_{bs} (v_1,\sigma) & \text{if } m_1, m_2 \in I^+,\ v_2 \models m_2 \wedge r,\ v_1 \models m_1
\end{array}$$

The set of entry states $\mathbb{E}_b$ is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$, where ϵ denotes the empty sequence.

Example 2. Consider the flow graph from Example 1. Because of possible unbounded recursion, it induces an infinite-state behaviour. One example execution of the program is represented by the following path from an initial to a final configuration:

$$\begin{array}{l}
(v_0,\epsilon) \xrightarrow{\tau}_{bs} (v_1,\epsilon) \xrightarrow{\tau}_{bs} (v_2,\epsilon) \xrightarrow{\text{even call odd}}_{bs} (v_5, v_3) \xrightarrow{\tau}_{bs} (v_6, v_3) \xrightarrow{\tau}_{bs} \\
(v_7, v_3) \xrightarrow{\text{odd call even}}_{bs} (v_0, v_9 \cdot v_3) \xrightarrow{\tau}_{bs} (v_1, v_9 \cdot v_3) \xrightarrow{\tau}_{bs} \\
(v_4, v_9 \cdot v_3) \xrightarrow{\text{even ret odd}}_{bs} (v_9, v_3) \xrightarrow{\text{odd ret even}}_{bs} (v_3, \epsilon)
\end{array}$$

Basic flow graph behaviour can be viewed as the behaviour of a pushdown automaton (PDA). Thus, behavioural properties can be verified using PDA model checking (see [5] for a survey of verification techniques for infinite-state systems). Notice further that for basic flow graphs, structural simulation indeed implies behavioural simulation (thus (3) holds), see [11].

2.4 Properties over Flow Graphs

As property specification language, we use a fragment of the modal μ-calculus [14] with boxes and greatest fixed-points only. A variety of useful safety properties of program control flow structure and behaviour are expressible in this fragment, as illustrated in our earlier work [11]. Let L be a set of labels, A a set of atomic propositions, and V a set of propositional variables.

Definition 6. (Logic) *The formulae of our* logic *are inductively defined by:* $\phi ::= p \mid \neg p \mid X \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid [a]\,\phi \mid \nu X.\phi$, *where* $p \in A$, $a \in L$ *and* $X \in V$.

Satisfaction of the logic is defined in terms of the general notion of specification in the standard way [14]. We use $\models_s$ and $\models_b$ to denote instantiation at the structural and behavioural level, respectively: $\mathcal{G} \models_s \phi \Leftrightarrow \mathcal{G} \models \phi$, and $\mathcal{G} \models_b \phi \Leftrightarrow b(\mathcal{G}) \models \phi$.

Example 3. For the flow graph in the basic program model from Example 1, the structural formula $\nu X.\ [\texttt{even}]\, r \wedge [\texttt{odd}]\, r \wedge [\varepsilon]\, X$ expresses the property "on every path from a program entry node, the first encountered call edge leads to a return node", in effect specifying that the program is tail-recursive. The behavioural formula $\neg\texttt{even} \vee \nu X.\ [\texttt{even call even}]\,\mathsf{ff} \wedge [\tau]\, X$ expresses the property "in every program execution that starts in method `even`, the first call is not to method `even` itself".

Due to the close correspondence between logical satisfaction and simulation, this logic is particularly suited for our compositional verification technique: there exists a mapping χ from finite specifications to formulae, and a mapping (maximal model construction) θ from formulae to finite specifications, such that for any specifications $\mathcal{S}, \mathcal{S}_1$ and finite $\mathcal{S}_2$ (see [11, Ths. 8, 15]):

$$\mathcal{S}_1 \leq \mathcal{S}_2 \Leftrightarrow \mathcal{S}_1 \models \chi(\mathcal{S}_2) \text{ and } \mathcal{S} \models \phi \Leftrightarrow \mathcal{S} \leq \theta(\phi) \tag{4}$$

2.5 Interface Characterisation

As mentioned above, our compositional verification technique is based on the construction of maximal models. However, for a given flow graph property, the maximal model does not necessarily correspond to a legal flow graph structure. Still, if for an interface I we can formulate a *characteristic formula* that precisely defines all legal flow structures with interface I, then we can use this formula to constrain maximal models to legal flow graph structures. Concretely, if σ_I is the characteristic formula for interface I, then the *maximal flow graph* for a property σ is defined as the maximal model (over labels and atomic propositions as induced by I) of the property $\sigma \wedge \sigma_I$. This describes a legal flow graph structure with interface I, simulating all other flow graphs with interface I, satisfying σ. Thus, for any instantiation of the general definition of flow graphs, to be able to apply our compositional verification principle, we need to define a formula σ_I that characterises all flow graphs with interface I, *i.e.*:

$$\mathcal{S} \models \sigma_I \Leftrightarrow \mathcal{R}(\mathcal{S}) : I \tag{5}$$

Basic Program Model. In the basic program model, flow graphs with interface I are models over $I^- \cup \{\varepsilon\}$ and $I^+ \cup \{r\}$ that can be characterised by the following formula ([11, Th. 31]), essentially specifying that every state is labelled by a unique method name that is preserved along edges:

$$\sigma_I = \bigvee_{m \in I^+} \nu X.P_m \wedge [I^-, \varepsilon]\, X \qquad P_m = m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m'$$

2.6 Compositional Verification

We can show that compositional verification principle (1) is sound and complete for any instantiation of flow graphs, provided that: *(i)* the notions of method specification and flow graph behaviour are defined as instances of the general notion of specification, *(ii)* structural simulation implies behavioural simulation (property (3)), and *(iii)* flow graphs with interface I can be characterised logically (property (5)). Together with properties (2) and (4), these are sufficient to prove soundness and completeness of the rule (see [11] for a detailed proof). The compositional verification principle applies to the basic program model, as shown in [11].

2.7 A Tool Set for Compositional Verification

In previous work [11], we implemented a tool set to support our compositional verification method in the context of the basic program model presented in Section 2.2. Figure 2 gives a general overview of its architecture.

For each component, we have as input either an implementation (in Java bytecode), or a structural property restricting its possible implementations and an interface specifying the provided and required methods. If we are given the code of the implementation, we use the Program Analyser to extract a flow graph

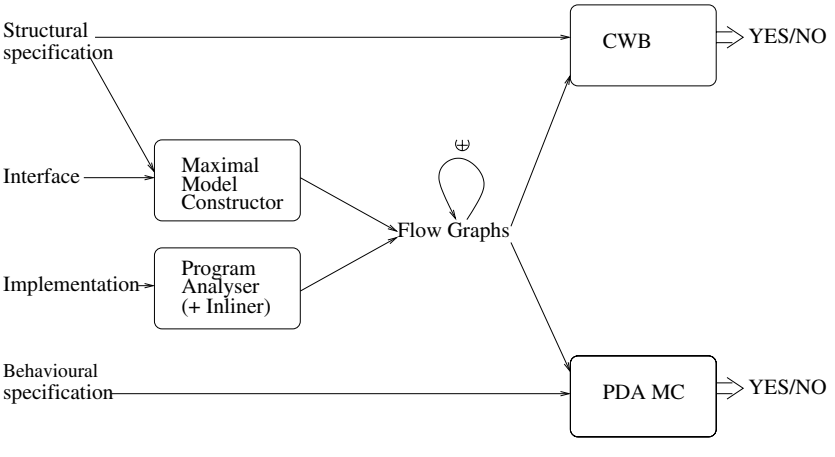


Fig. 2. Tool Set for Compositional Verification

(and if necessary, we use the Inliner to abstract the flow graph to public methods, *i.e.*, methods mentioned in the interface, only [11]). If we are given a structural property, we construct a maximal flow graph as described in Section 2.5 using the Maximal Model Constructor. Composition $\uplus$ of the the resulting flow graphs basically amounts to a concatenation of the textual graph representations. The tool set also implements translations of flow graphs into models which serve as input for different model checkers. In order to check structural properties, we exploit the fact that flow graphs can be viewed as finite Kripke structures, and convert flow graphs to CCS models. Since structural properties are μ–calculus formulae, the verification can then be done using standard model checking tools such as the Concurrency Workbench (CWB) [6]. To verify that a composed system respects a behavioural safety property, we view the behaviour of a flow graph as an infinite state model generated by a Pushdown Automaton (PDA), and apply PDA model checking. We are not aware of an efficient, off-the-shelf model checker for (alternation–free) modal μ–calculus properties of PDAs. We are currently developing one ourselves.

The extensions to the Program Analyser for handling exceptional and multi-threaded control flow are described in the following sections. Extending the Maximal Model Constructor, Inliner and the translation into CCS and PDA models is straightforward, and not discussed further.

The tool set has been evaluated on the PACAP case study [4], an electronic purse developed for smart cards. In PACAP, a smart card may contain one purse applet and several loyalty applets, which interact to exchange information. The case study describes a potential "bad scenario" in terms of an illicit interaction involving the purse applet and the loyalty applets, one of which is malicious. Goal of the verification, presented in detail in [11], is to ensure the absence of such illicit interactions for the given implementations of the purse and loyalties.

3 Instantiation: Exceptional Control Flow

As a first example of how the compositional verification principle can be instantiated to richer program models, we present an instantiation with exceptions. For this, we take $\mathcal{C}ontr$ to be $\mathcal{E}xcp$, an infinite set of exception names, and we define method specifications over $M \subseteq \mathcal{M}eth$ and $E \subseteq \mathcal{E}xcp$.

In a flow graph with exceptions, a control point may be tagged with an exception: the state is said to be exceptional if the current control point is tagged with an exception (*cf.* having an exception at the top of the operand stack [16]). Model extraction from actual bytecode models every instruction that might raise an exception with several transfer edges, one leading to a normal and the others leading to exceptional control points (for all possible exceptions). Explicit throw statements are modelled as internal transfer edges that always lead to an exceptional point. Catch statements are implicit: they are modelled as internal transfer from an exceptional to a normal control point.

At behavioural level, the main difference with the basic model is that the decision in which control point execution resumes after completion of a method

call is postponed to the time of return, depending on whether the method call returns normally, or with an exception. Model extraction for a method that may terminate with an exception produces multiple edges labelled with this method, ending in control points tagged with exceptions, in addition to an edge that ends in a normal control point. When a method is called, the set of all possible return points (exceptional and normal) is pushed on the call stack (instead of a single one), so that the appropriate control point can be selected upon return.

Below, we instantiate the compositional verification principle for flow graphs with exceptions in such a way that conditions *(i)-(iii)* from Section 2.6 are met. In particular, we define structure and behaviour appropriately. We also discuss how model extraction is adapted, and we give typical example properties that refer to the exceptional structure or behaviour of a flow graph.

3.1 Program Model with Exceptions

As mentioned above, we instantiate $\mathcal{C}ontr$ with $\mathcal{E}xcp$. Interfaces of flow graphs with exceptions are thus of the form (I^+, I^-, E), where $E \subseteq \mathcal{E}xcp$. We use $I^{\mathcal{E}}$ to extract the exception component from the interface.

Method specifications are very similar to method specifications in the basic program model, except that we add exceptions as atomic propositions.

Definition 7. (Method Specification with Exceptions) *A* flow graph with exceptions *for* $m \in \mathcal{M}eth$ *over sets* $M \subseteq \mathcal{M}eth$ *and* $E \subseteq \mathcal{E}xcp$ *is a finite model* $\mathcal{M}_m = (V_m, L_m, \rightarrow_m, A_m, \lambda_m)$ *with* V_m *the set of control nodes of* m, $L_m = M \cup \{\varepsilon\}$, $A_m = \{m, r\} \cup E$, $m \in \lambda_m(v)$ *for all* $v \in V_m$, *and for all* $e, e' \in E$, *if* $\{e, e'\} \subseteq \lambda_m(v)$ *then* $e = e'$, i.e., *each control point is tagged with at most one exception. A* method specification with exceptions *for* $m \in \mathcal{M}eth$ *over* M *and* E *is a specification* $(\mathcal{M}_m, \mathbb{E}_m)$ *s.t.* $\mathcal{M}_m$ *is a flow graph with exceptions for* m *over* M *and* E *and* $\mathbb{E}_m \subseteq V_M$ *a non-empty set of* entry points *of* m.

We use the following abbreviation: $v \models E \Leftrightarrow \exists e \in E. v \models e$. Method specifications with exceptions have to satisfy two *wellformedness* constraints: (1) entry nodes are not exceptional: $\forall v \in \mathbb{E}_m. v \not\models I^{\mathcal{E}}$; and (2) all outgoing edges from exceptional control points are internal transfer edges ending in a normal control point: $\forall v, v' \in V, e \in I^{\mathcal{E}}, l \in L_m. v \models e \wedge v \xrightarrow{l} v' \Rightarrow l = \varepsilon \wedge v' \not\models I^{\mathcal{E}}$. The second constraint is not strictly necessary, but keeps the behaviour of flow graphs clean: catching an exception always results in a normal state in the same method. Throughout, we will assume all method specifications to be wellformed.

3.2 Extracting Flow Graphs with Exceptions from Java Classes

We extended the Program Analyser to handle exceptions. Explicit throw statements give rise to internal transfer edges ending in an appropriately labelled exceptional control point. All other instructions that might raise an exception (such as accessing a reference, which can lead to a NullPointerException) are modelled by a choice: the current control point has multiple outgoing edges labelled ε, one ending in a normal control point and all others ending in appropriate

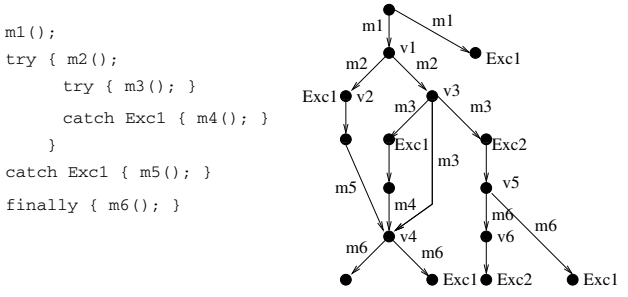


Fig. 3. Example extraction for a try-catch-finally statement

exceptional control points. To model method invocations, edges are labelled either ε, modelling the case that the invocation instruction raises an exception, or with the method name. At most one of the edges labelled with the method name ends in a normal control point, modelling normal termination of the method, all others lead to an exceptional control point, corresponding to exceptional returns from the method. The exceptional control points are either tagged with an exception listed in the method's throw clause, or with a runtime exception that can be thrown (and not caught) in the method. The analysis which exceptions might be returned by a method is transitive *w.r.t.* the call graph.

To illustrate how PA extracts a flow graph from a `try-catch-finally` block, Figure 3 shows an example code fragment[3] together with the corresponding flow graph. We assume that `Exc1` and `Exc2` are the only exceptions; `m1`, `m2` and `m3` and `m6` can throw `Exc1`, while `m3` can also throw `Exc2`. (For presentation purposes, some nodes are named.) A `try-catch` is modelled by branches in the control flow: each instruction in the try-block that could raise an exception has an outgoing edge to an exceptional control point (*e.g.*, the call to `m2` in `v1` can lead to normal point `v3`, or to exception point `v2`). If the exception is handled by one of the catch clauses, the only outgoing edge from this point leads to the control flow of the corresponding clause. For example, in `v2`, the exception is caught by the outer catch clause, leading to a call of `m5`. All edges that correspond to normal termination of the `try-catch` (*i.e.,* termination of the try-block, and termination of all catch-clauses) lead to the same control point, where the flow graph modelling the next instruction starts. If the `try-catch` block is followed by a `finally`-clause, at each possible exit of the `try-catch` block (*e.g.,* nodes `v4` and `v5` in Figure 3), the graph extracted for the `finally` clause is inserted. In case the `try-catch` block ended with an exception, the exception is saved until the end nodes of the graph of the `finally` clause, thus the internal nodes of the `finally` graph are not tagged with this exception. However if an end node of the `finally` graph is normal, an edge is added to rethrow the exception. For example, if the call to `m6` in `v5` ends normally in `v6`, then `Exc2` is re-thrown.

[3] For illustrative purposes, the extraction is described in terms of source code, however the actual implementation works on bytecode.

The end node of a `finally` clause can thus be either normal, tagged with an exception thrown in the `finally` block or with the exception inherited from the `try-catch` block (in case no exception is thrown by the `finally` block itself).

In order to see the results of graph extraction on a realistic piece of software, we analysed a simulation application built on top of the JavaSim library, a tool for building discrete event process-based simulation[4]. We considered 140 types of exceptions, checked as well as unchecked, all subtypes of class `Exception`. The exceptional control flow graph includes 55 methods in 14 classes (approximately 640 lines of code), of which 7 classes belong to the JavaSim library. On a Pentium4 2.2GHz computer with 512MB memory pool, the call graph construction takes 3 minutes, and can be decreased substantially by instrumenting Soot to prevent the analysis of Java API methods. It takes 1,5 seconds to create the control flow graph, which contains 1450 nodes and 1466 edges.

3.3 Flow Graph Behaviour with Exceptions

Modelling the behaviour of flow graphs with exceptions requires a different use of the call stack than in the basic program model. In that model, the return point is determined and pushed on the call stack at the time the method is called. But when modelling exceptional behaviour, it cannot be predicted at call time whether termination will be normal or exceptional. Therefore, the call transition pushes the set of all possible return points on the call stack, and the return transition selects the appropriate one, *i.e.*, with the matching exception (if any). In addition, we introduce transition labels $\mathsf{throw}\, e$ and $\mathsf{catch}\, e$; this makes raising and recovering from exceptions observable for specification purposes.

Definition 8 (Behaviour with Exceptions). *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed flow graph with exceptions such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* behaviour *of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{be}, A_b, \lambda_b)$ s.t. $S_b \in V \times (\mathcal{P}(V) \backslash \{\varnothing\})^*$,* i.e., *states are pairs of control points and stacks of non-empty sets of nodes, $L_b = \{m_1\ l\ m_2 \mid l \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\} \cup \{l\ e \mid l \in \{\mathsf{throw}, \mathsf{catch}\},\ e \in I^{\mathcal{E}}\}$, $A_b = A$, $\lambda_b((v, \sigma)) = \lambda(v)$ and $\rightarrow_{be}$ is defined as follows:*

$$
\begin{array}{rll}
\text{[transfer]} & (v,\sigma) \xrightarrow{\tau}_{be} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r,\ v \not\models I^{\mathcal{E}},\ v' \not\models I^{\mathcal{E}} \\
\text{[call]} & (v_1,\sigma) \xrightarrow{m_1\ \mathsf{call}\ m_2}_{be} (v_2, V \cdot \sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \models \neg r,\ v_1 \not\models I^{\mathcal{E}},\ v_2 \models m_2, \\
 & & \quad v_2 \in \mathbb{E},\ V = \{v \mid v_1 \xrightarrow{m_2}_{m_1} v\},\ V \neq \varnothing \\
\text{[return]} & (v_2, V \cdot \sigma) \xrightarrow{m_2\ \mathsf{ret}\ m_1}_{be} (v_1,\sigma) & \text{if } m_1, m_2 \in I^+,\ v_1 \models m_1,\ v_2 \models m_2 \wedge r, \\
 & & \quad v_1 \in V,\ \forall e \in I^{\mathcal{E}}.v_1 \models e \Leftrightarrow v_2 \models e \\
\text{[throw]} & (v,\sigma) \xrightarrow{\mathsf{throw}\ e}_{be} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r,\ v' \models e \\
\text{[catch]} & (v,\sigma) \xrightarrow{\mathsf{catch}\ e}_{be} (v',\sigma) & \text{if } m \in I^+,\ v \xrightarrow{\varepsilon}_m v',\ v \models \neg r \wedge e
\end{array}
$$

The set of initial states $\mathbb{E}_b$ is defined by $\mathbb{E}_b = \mathbb{E} \times \{\epsilon\}$.

[4] Available via the JavaSim homepage: `http://javasim.ncl.ac.uk`.

As for the basic model, the behaviour of a flow graph with exceptions is the behaviour of a PDA, and hence PDA model checkers can again be used for verification of behavioural properties. Since there is a close correspondence between flow graph structure and behaviour, structural simulation between flow graphs with exceptions implies their behavioural simulation (thus property (3) holds).

Theorem 1. *Let $\mathcal{G}_1$ and $\mathcal{G}_2$ be flow graphs with exceptions. If $\mathcal{G}_1 \leq_s \mathcal{G}_2$ then $\mathcal{G}_1 \leq_b \mathcal{G}_2$.*

Proof. Let R be a structural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. Define relation R_b by (where $|\sigma|$ denotes the length of σ, and $\sigma(i)$ the i^{th} element in σ):

$$(v,\sigma)R_b(v',\sigma') \Leftrightarrow vRv' \wedge |\sigma| = |\sigma'| \wedge \forall i < |\sigma|.\forall w \in \sigma(i).\exists w' \in \sigma'(i).wRw'$$

It is easy to check that R_b is a behavioural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. □

3.4 Properties over Flow Graphs with Exceptions

Modelling exceptional control flow of flow graphs not only allows to better approximate their behaviour, it also allows to express and verify properties that are related to exceptions (both at structural and at behavioural level). Typical properties of a flow graph with exceptions $\mathcal{G} : I$ expressible in our logic are:

- Exception $e \in I^{\mathcal{E}}$ is never thrown*:* $\nu X.\neg e \wedge [-]\, X$ (where $[K]\phi$ abbreviates $\bigwedge_{a\in K}[a]\phi$ and '$-$' stands for L). Notice that this property can be expressed both at structural and at behavioural level (but with a slightly different meaning: at the behavioural level, recursion is taken into account, thus certain control points might never be reachable).
- Exception $e \in I^{\mathcal{E}}$ is always caught within the method where it is thrown: $\nu X.(\neg e \vee \neg r) \wedge [-]\, X$ (again, this property can be expressed both at structural and behavioural level).
- After exception $e \in I^{\mathcal{E}}$ is thrown, the first method that can be called is the (state-restoring) method $n \in I^{+}$: $\nu X.(\neg e \vee \nu Y.\, [M \setminus \{n\}]\, \mathsf{ff} \wedge [\varepsilon]\, Y) \wedge [-]\, X$.

It is natural to handle exceptions locally. Hence, in a compositional verification setting, global behavioural specifications would typically not mention throwing and catching of exceptions; these labels can instead be relabelled into silent τ-transitions.

The tool set has also been extended to translate control flow graphs with exceptions into CCS models. This has been used to produce the CCS model corresponding to the graph extracted for the simulation application described at the end of Section 3.2. Then, we used the Concurrency Workbench to verify various local properties of the application. For instance, we checked whether exceptions are caught locally, i.e., within the method. For the `finalize()` method of JavaSim's `SimulationProcess` class, shown in Figure 4, and a particular exception e, the property `finalize` $\Rightarrow \nu X.(\neg(e \wedge r)) \wedge [-]\, X$ specifies that exceptions of type e are caught locally. The instructions in the `finalize()` method that may raise an exception are the calls to the virtual method `idle()`, the static

```
public void finalize () {
    if (!Terminated) {
          Terminated = true; Passivated = true;
          wakeuptime = SimulationProcess.Never;
          if (!idle()) Scheduler.unschedule(this);
          if (this == SimulationProcess.Current) {
              try { Scheduler.schedule(); }
              catch (SimulationException e) { } }
          SimulationProcess.allProcesses.Remove(this); }}
```

Fig. 4. The `finalize()` method of JavaSim's `SimulationProcess` class

methods `unschedule()`, `schedule()`, `Remove()` and accesses to the fields `Never`, and `Current`. All but one of these instructions raise only the `NullPointerException`: the call to method `schedule()` might raise `NullPointerException` and `SimulationException`, an application-defined exception. Model checking the property succeeded for all exceptions e except for `NullPointerException`, showing that not all exceptions are caught locally.

3.5 Interface Characterisation of Flow Graphs with Exceptions

Given an interface for a flow graph with exceptions I, we can characterise the flow graphs with this interface by the formula σ_I, essentially stating that any initial control point is normal, and after a transition, either the control point is normal again, or we are in an exceptional point, where all outgoing edges are internal transfer edges, leading to a normal control point:

$$\sigma_I = \bigvee_{m \in I^+} (\nu X.\, P_m \wedge \bigwedge_{e \in I^\varepsilon} \neg e \wedge [I^-, \varepsilon]\,(X \vee (\bigwedge_{m \in I^+} [m]\,\mathsf{ff} \wedge P_m \wedge \bigvee_{e \in I^\varepsilon} P_e \wedge [\varepsilon]X)))$$

$$P_m = m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m' \qquad P_e = e \wedge \bigwedge_{e' \in I^\varepsilon \setminus \{e\}} \neg e'$$

The following result tells us that σ_I indeed characterises all flow graphs with exceptions with interface I, thus (5) holds.

Theorem 2. *Let I be an interface for flow graphs with exceptions. For any specification $\mathcal{S} = (\mathcal{M}, \mathbb{E})$ over labels $L = I^- \cup \{\varepsilon\}$ and atomic propositions $A = I^+ \cup \{r\} \cup E$ we have (where $\mathcal{R}$ denotes the reachable part of a specification, as defined on page 150): $\mathcal{S} \models_s \sigma_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$.*

Proof. Similar to the proof of Theorem 31 in [11]. □

Because of this result and Theorem 1 the compositional verification principle (1) also applies to flow graphs with exceptions.

4 Instantiation: Multi-threaded Control Flow

As a second example, we instantiate the generic program model with multi-threaded control flow. In this case, the set of control values consists of lock and

thread names, *i.e.,* $\mathcal{C}ontr = \mathcal{L}ock \times \mathcal{T}id$, where $\mathcal{L}ock$ and $\mathcal{T}id$ are infinite sets of lock and thread names, respectively. Given an interface I, we use $I^{\mathcal{L}}$ and $I^{\mathcal{T}}$ to extract the legal lock and thread names, respectively.

Our program model supports all basic thread constructs as provided by Java: thread creation, monitors, a wait-notify mechanism, and the possibility to join a thread (*i.e.*, wait for its completion). The behaviour of this instantiation extends the behaviour of the basic program model, by maintaining a configuration for each thread. We assume that (the interleaving behaviours of) programs do not contain data races and thus, by virtue of the Java Memory Model [17], we can assume an interleaving semantics. Notice that the program model described in this section can be easily combined with the program model described above into a single program model with multi-threading and exceptions.

4.1 Program Model with Multi-threading

To define method specifications for multi-threaded programs, we introduce edge labels that correspond to the instructions specific to multi-threading. Following the Java semantics, the body of a method will be executed sequentially, possibly starting new threads, interleaved with other threads. Let $L_{M,L,T}$ abbreviate the set of labels $M \cup \{\varepsilon\} \cup \{c\, l \mid c \in \{\mathsf{lock}, \mathsf{unlock}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}\}, l \in L\} \cup \{\mathsf{spawn}\ t\ \mathsf{with}\ m \mid t \in T, m \in M\} \cup \{\mathsf{join}\ t \mid t \in T\}$.

Definition 9. (Method Specification with Multi-threading) *A* flow graph with multi-threaded control flow *for $m \in \mathcal{M}eth$ over sets $M \subseteq \mathcal{M}eth$, $L \subseteq \mathcal{L}ock$ and $T \subseteq \mathcal{T}id$ is a finite model $\mathcal{M}_m = (V_m, L_{M,L,T}, \rightarrow_m, A_m, \lambda_m)$ with V_m the set of control nodes of m, $A_m = \{m, r\}$, and $m \in \lambda_m(v)$ for all $v \in V_m$. A* method specification with multi-threaded control flow *for $m \in \mathcal{M}eth$ over M, L and T is a specification $(\mathcal{M}_m, \mathbb{E}_m)$ with $\mathcal{M}_m$ a method graph with multi-threaded control flow for m over M, L and T, and $\mathbb{E}_m \subseteq V_m$ a non-empty set of entry points of m.*

4.2 Extracting Flow Graphs from Multi-threaded Java Classes

To extend the Program Analyser to multi-threaded classes, we generate edges with appropriate labels for all (non-deprecated) Java primitives and native methods related to concurrency, with the exception of the timed wait and the interrupt mechanism. For instance, calling the `start` (or `fork`) method on a thread object, is modelled by an edge labelled spawn, while a call to `join` leads to an edge labelled join. Special care is taken for calls to synchronized methods: they are preceded and followed by edges labelled lock and unlock on the appropriate object, *i.e.*, the synchronisation is made explicit.

Special care has to be taken to ensure that the extracted sets of thread and lock names are finite. For threads, a safe over-approximation is to use the declared class name of the thread as thread name in the model. Using a more precise analysis can help to distinguish different threads that are instances of the same class. For locks, abstracting with the class name might under-approximate the program behaviour. To overcome this problem, we require that the program has only a finite number of lock objects with the same class name.

Table 1. Transition rules $\rightarrow_{bm}$ for multi-threaded behaviour

$$
\begin{array}{rll}
\text{[exec.]} & (\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma(t{:=}(v', \sigma')), \mathsf{L}, \mathsf{W}) & \text{if } t \notin \mathsf{W},\ \Sigma(t) \xrightarrow{a}_{bs} (v', \sigma') \\
\text{[coord.]} & (\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma(t{:=}(v', \sigma)), \mathsf{L}', \mathsf{W}') & \text{if } \Sigma(t) = (v, \sigma), t \notin \mathsf{W},\ v \xrightarrow{a}_{m} v', \\
 & & m \in I^+,\ (\mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{c} (\mathsf{L}', \mathsf{W}') \\
\text{[resume]} & (\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,\ \mathsf{resume}\ l)}_{bm} (\Sigma, \mathsf{L}', \mathsf{W}') & \text{if } (t, n, \mathsf{tt}) \in \mathsf{W}(l),\ \mathsf{L}' = \mathsf{L}(l{:=}(t, n)), \\
 & & \mathsf{L}(l) = \bot,\ \mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \backslash (t, n, \mathsf{tt})) \\
\text{[thr.-ops.]} & (\Sigma, \mathsf{L}, \mathsf{W}) \xrightarrow{(t,a)}_{bm} (\Sigma'(t{:=}((v', \sigma)), \mathsf{L}, \mathsf{W}) & \text{if } \Sigma(t) = (v, \sigma), t \notin \mathsf{W}, \\
 & & v \xrightarrow{a}_{m} v'\ m \in I^+,\ \Sigma \xrightarrow{a}_{t} \Sigma'
\end{array}
$$

4.3 Flow Graph Behaviour with Multi-threading

The behaviour specification follows closely the Java Specification [16]. Instead of a single call stack, we maintain a map from thread identifiers to configurations (*i.e.,* control point and call stack). If a thread is not active, it maps to $\bot$. Further, the state space also contains a *lock map* and a *wait map*. The lock map returns for each lock the identity of the thread holding the lock and the lock counter (*i.e.,* how many times the lock is held, necessary to correctly model the reentrant locking behaviour of Java). The wait map returns for each lock the set of threads that are waiting for it, the number of times the thread was holding the lock when it started waiting, and a flag whether the thread has been notified. This ensures that the thread resumes in the exact same state as when it issued a wait, thus making sure a correct number of unlocks is necessary to release the lock. We explicitly require that if a thread is waiting for a lock, its state is active.

We assume execution starts in a special thread called main, and that any closed flow graph contains such a thread. Labels and atomic propositions are paired with thread identifiers. Further, we introduce the atomic proposition $\mathsf{haslock}(t, l)$ to hold in any state where thread t holds lock l.

Definition 10. (Behaviour with Multi-threading) *Let $\mathcal{G} = (\mathcal{M}, \mathbb{E}) : I$ be a closed multi-threaded flow graph such that $\mathcal{M} = (V, L, \rightarrow, A, \lambda)$. The* multi-threaded behaviour *of $\mathcal{G}$ is described by the specification $b(\mathcal{G}) = (\mathcal{M}_b, \mathbb{E}_b)$, where $\mathcal{M}_b = (S_b, L_b, \rightarrow_{bm}, A_b, \lambda_b)$ is defined as follows:*

- $S_b = \{s \in (I^{\mathcal{T}} \rightarrow (V \times V^*)_\bot) \times (I^{\mathcal{L}} \rightarrow (I^{\mathcal{T}} \times \mathbb{N})_\bot) \times (I^{\mathcal{L}} \rightarrow \mathcal{P}(I^{\mathcal{T}} \times \mathbb{N} \times \mathbb{B})) \mid \forall l, t, n, b.(t, n, b) \in \pi_3(s)(l) \Rightarrow \pi_1(s)(t) \neq \bot\}$,
- $L_b = T \times (\{m_1\ c\ m_2 \mid c \in \{\mathsf{call}, \mathsf{ret}\},\ m_1, m_2 \in I^+\} \cup \{\tau\} \cup \{c\, l \mid c \in \{\mathsf{lock}, \mathsf{unlock}, \mathsf{wait}, \mathsf{notify}, \mathsf{notifyAll}, \mathsf{resume}\}, l \in I^{\mathcal{L}}\} \cup \{\mathsf{spawn}\ t\ \mathsf{with}\ m \mid t \in I^{\mathcal{T}}, m \in I^+\} \cup \{\mathsf{join}\ t \mid t \in I^{\mathcal{T}}\})$,
- $\rightarrow_{bm}$ *is defined in Table 1*[5] *(using auxiliary rules $\rightarrow_c$ and $\rightarrow_t$ of Table 2) ,*
- $A_b = (T \times A) \cup \{\mathsf{haslock}(t, l) \mid t \in I^{\mathcal{T}}, l \in I^{\mathcal{L}}\}$, *and*
- $\lambda_b(s) = \{(t, p) \mid t \in I^{\mathcal{T}} \wedge \pi_1(s)(t) \neq \bot \wedge p \in \lambda(\pi_1(\pi_1(s)(t)))\} \cup \{\mathsf{haslock}(t, l) \mid \pi_2(s)(l) \neq \bot \wedge \pi_1(\pi_2(s)(l)) = t\}$.

[5] We abbreviate $\exists n, b, l.(t, n, b) \in \mathsf{W}(l)$ as $t \in \mathsf{W}$. We use $f(i{:=}x)$ to denote function update. Further, $\Sigma(i) = (v, \sigma)$ implicitly implies that $\Sigma(i) \neq \bot$.

Table 2. Auxiliary transition rules $\rightarrow_c$ and $\rightarrow_t$

$$
\begin{array}{rll}
\text{[lock]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{lock}\ l)}_c (\mathsf{L}',\mathsf{W}) & \text{if } \mathsf{L}(l) = \bot,\ \mathsf{L}' = \mathsf{L}(l{:=}(t,1)) \\
\text{[re-lock]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{lock}\ l)}_c (\mathsf{L}',\mathsf{W}) & \text{if } \mathsf{L}(l) = (t,n),\ \mathsf{L}' = \mathsf{L}(l{:=}(t,n)) \\
\text{[unlock]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{unlock}\ l)}_c (\mathsf{L}',\mathsf{W}) & \text{if } \mathsf{L}' = \mathsf{L}(l{:=}(\mathsf{L}(l) = \bot \vee \mathsf{L}(l) = (t,1))\,?\ \bot : (\pi_1(\mathsf{L}(l)), \pi_2(\mathsf{L}(l)) - 1)) \\
\text{[wait]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{wait}\ l)}_c (\mathsf{L}',\mathsf{W}') & \text{if } \mathsf{L}(l) = (t,n),\ \mathsf{L}' = \mathsf{L}(l{:=}\bot),\ \mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \cup \{(t,n,\mathsf{ff})\}) \\
\text{[notify]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notify}\ l)}_c (\mathsf{L},\mathsf{W}') & \text{if } \mathsf{L}(l) = (t,n), (t',n,\mathsf{ff}) \in \mathsf{W}(l),\ \mathsf{W}' = \mathsf{W}(l{:=}\mathsf{W}(l) \backslash \{(t',n,\mathsf{ff})\} \cup \{(t',n,\mathsf{tt})\}) \\
\text{[notify-cont]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notify}\ l)}_c (\mathsf{L},\mathsf{W}) & \text{if } \mathsf{L}(l) = (t,n), \forall t'.(t',n,\mathsf{ff}) \notin \mathsf{W}(l) \\
\text{[notifyAll]} & (\mathsf{L},\mathsf{W}) \xrightarrow{(t,\mathsf{notifyAll}\ l)}_c (\mathsf{L},\mathsf{W}') & \text{if } \mathsf{L}(l) = (t,n),\ \mathsf{W}' = \mathsf{W}(l{:=}\{(t',n,\mathsf{tt}) \mid (t',n,r) \in \mathsf{W}(l)\}) \\
\text{[spawn]} & \Sigma \xrightarrow{\mathsf{spawn}\ t'\ \mathsf{with}\ m'}_t \Sigma' & \text{if } \Sigma(t') = \bot,\ m' \in I^+,\ v'' \in E,\ v'' \models m',\ \Sigma' = \Sigma(t'{:=}(v'',\epsilon)) \\
\text{[join]} & \Sigma \xrightarrow{\mathsf{join}\ t'}_t \Sigma & \text{if } \Sigma(t') = (v'',\epsilon),\ v'' \models r
\end{array}
$$

The set of initial states $\mathbb{E}_b$ *is defined as* $\mathbb{E}_b = \{(\Sigma_I^v, \lambda l.\ \bot, \lambda l.\varnothing) \mid v \in \mathbb{E}\}$ *where* $\Sigma_I^v(\mathsf{main}) = (v, \epsilon, \bot)$ *and* $\Sigma_I^v(t) = \bot$ *for all* $t \in I^{\mathcal{T}}$.

The transition rules should be understood as follows. Rule [exec.] lifts the standard rules for sequential flow graphs ($\rightarrow_{bs}$, Def. 5) to the multi-threaded case. Rule [coord.] models the coordination of threads via locks, *i.e.,* (un)lock, wait, and notify(All): the current thread changes control point if the lock and wait map can be updated appropriately, as defined by the auxiliary transition rules $\rightarrow_c$ (see [12] for more explanation). Rule [thr.-ops.] models creating and joining a thread using the auxiliary transition rules $\rightarrow_t$ (see also [12]). Finally, rule [resume] handles the case where an thread is waiting on an object, has been notified, and now continues execution.

Also in the case of multi-threaded flow graphs, there is a direct correspondence between flow graph structure and behaviour, and thus structural simulation implies behavioural simulation.

Theorem 3. *Let* $\mathcal{G}_1$ *and* $\mathcal{G}_2$ *be flow graphs with multi-threading. If* $\mathcal{G}_1 \leq_s \mathcal{G}_2$ *then* $\mathcal{G}_1 \leq_b \mathcal{G}_2$.

Proof. Let R be a structural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. Define

$$
\begin{array}{l}
(\Sigma, \mathsf{L}, \mathsf{W}) R_b (\Sigma', \mathsf{L}', \mathsf{W}') \Leftrightarrow \\
(\forall t \in T.\, \mathsf{if}\ \Sigma(t) = (v, \sigma) \\
\qquad \mathsf{then}\ \Sigma'(t) = (v', \sigma') \wedge v R v' \wedge |\sigma| = |\sigma'| \wedge \forall i.i < |\sigma|.\sigma(i) R \sigma'(i) \\
\qquad \mathsf{else}\ \Sigma'(t) = \bot) \wedge \mathsf{L} = \mathsf{L}' \wedge \mathsf{W} = \mathsf{W}'
\end{array}
$$

It is easy to check that R_b is a behavioural simulation between $\mathcal{G}_1$ and $\mathcal{G}_2$. □

4.4 Properties over Flow Graphs with Multi-threading

The instantiation of the generic flow graph model with multi-threaded control flow allows us to express properties that are related to the multi-threaded character of the flow graph. Given a flow graph $\mathcal{G} : I$ with multi-threaded control flow, typical (behavioural) properties expressible in our logic are:

- Method $m \in I^+$ can only be called by thread t, if t has lock l: $\nu X. \bigwedge t \in I^{\mathcal{T}} (\mathsf{haslock}(t, l) \vee \bigwedge_{m' \in I^+} [(t, m'\ \mathsf{call}\ m)]\,\mathsf{ff}) \wedge [-]\,X$. If method m is the only method accessing some data, this means that data is lock protected.
- Locks are acquired in a particular order, for example lock l_2 can only be acquired by a thread that already has lock l_1: $\nu X. \bigwedge_{t \in I^{\mathcal{T}}} (\mathsf{haslock}(t, l_1) \vee [(t, \mathsf{lock}\ l_2)]\,\mathsf{ff})] \wedge [-]\,X$. This guarantees absence of deadlocks by synchronisation (however, it does not guarantee absence of deadlocks, caused by the wait-notify mechanism, or by joining a non-terminating thread).
- No more than n threads are created in an application. This is an important resource property. Formally, this can be expressed as $MaxThr\ (n)$, inductively defined as follows:

$$MaxThr\ (1) = \nu X_1. \bigwedge_{m \in I^+, t \in I^{\mathcal{T}}} [\mathsf{spawn}\ t\ \mathsf{with}\ m]\,\mathsf{ff} \wedge [-]\,X_1$$
$$MaxThr\ (k+1) = \nu X_{k+1}. \bigwedge_{m \in I^+, t \in I^{\mathcal{T}}} [\mathsf{spawn}\ t\ \mathsf{with}\ m]\,MaxThr\ (k) \wedge [-]\,X_{k+1}$$

4.5 Interface Characterisation of Flow Graphs with Multi-threading

Given an interface for a flow graph with multi-threaded control flow I, the flow graphs with this interface can be characterised by the formula σ_I, where $L_{M,L,T}$ is as defined on Page 161:

$$\sigma_I = \bigvee_{m \in I^+} (\nu X. P_m \wedge \left[L_{I^-, I^{\mathcal{L}}, I^{\mathcal{T}}}\right] X) \qquad P_m = m \wedge \bigwedge_{m' \in I^+ \setminus \{m\}} \neg m$$

Theorem 4. *Let I be an interface for multi-threaded flow graphs. For any specification $\mathcal{S} = (\mathcal{M}, E)$ over labels $I^- \cup \{\varepsilon\} \cup L_{M,L,T}$ and atomic propositions $A = I^+ \cup \{r\}$ we have : $\mathcal{S} \models_s \sigma_I$ if and only if $\mathcal{R}(\mathcal{S}) : I$.*

Proof. Similar to the proof of Theorem 31 in [11]. □

Thus, the compositional verification principle (1) also applies to flow graphs with multi-threading. However, applying the verification principle poses a problem to model checking, since the verification problem resulting from the second premise, $\theta_I(\phi) \uplus \mathcal{G}_2 \models_b \phi$, is not decidable in general for the case of pushdown systems with multiple stacks. This is a consequence of a basic undecidability result due to Ramalingam [22], which is related to the undecidability of the problem of emptiness of intersection of context-free languages. Hence, every such model checking algorithm must use an under- or over-approximation of the program behaviour. Different approaches have been proposed, see *e.g.,* [3,8,21]. It is future work to study whether and how these solutions can be integrated into our framework.

5 Conclusion

This paper discusses how a previously developed method for compositional verification of control-flow properties of sequential flow graphs with procedures can be adapted to richer program models. We present a generic program model, of which the original program model is an instantiation, and explicate the conditions under which the compositional verification principle is sound and complete. Two other example instantiations of this generic model are presented: with exceptional and with multi-threaded control flow. Also for these particular instantiations, the compositional verification principle holds (noting, however, that in the case of multi-threaded flow graphs we lose decidability due to a general undecidability result for pushdown systems with multiple stacks). The restrictions on the instantiations required to ensure soundness and completeness of the principle are not severe, and the resulting models are intuitive and standard – and can thus be used for other analyses as well. It is future work to study other possibilities to enrich the program model, for example by adding data (from finite domains), or access control information. We are currently adapting the tool set to handle multi-threaded models.

References

1. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M.: Analysis of recursive state machines. ACM TOPLAS 27, 786–818 (2005)
2. Besson, F., Jensen, T., Le Métayer, D., Thorn, T.: Model checking security properties of control flow graphs. J. of Computer Security 9(3), 217–250 (2001)
3. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. SIGPLAN Notes 38(1), 62–73 (2003)
4. Bretagne, E., El Marouani, A., Girard, P., Lanet, J.-L.: PACAP purse and loyalty specification. Technical Report V 0.4, Gemplus (2000)
5. Burkart, O., Caucal, D., Moller, F., Steffen, B.: Verification on infinite structures. In: Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.) Handbook of Process Algebra, pp. 545–623. North-Holland, Amsterdam (2000)
6. Cleaveland, R., Parrow, J., Steffen, B.: A semantics based verification tool for finite state systems. In: International Symposium on Protocol Specification, Testing and Verification, pp. 287–302. North-Holland Publishing Co., Amsterdam (1990)
7. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 489–503. Springer, Heidelberg (2006)
8. Esparza, J., Podelski, A.: Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In: Principles of programming languages (POPL 2000), pp. 1–11. ACM Press, New York (2000)
9. Grumberg, O., Long, D.: Model checking and modular verification. ACM TOPLAS 16(3), 843–871 (1994)
10. Gurov, D., Huisman, M.: Reducing behavioural to structural properties of programs with procedures. Technical Report TRITA-CSC-TCS 2007:3, KTH Royal Institute of Technology, Stockholm (2007)
11. Gurov, D., Huisman, M., Sprenger, C.: Compositional verification of sequential programs with procedures. Information and Computation 206(7), 840–868 (2008)

12. Huisman, M., Aktug, I., Gurov, D.: Flow graph behaviour for multi-threaded applications (2007), ftp://ftp-sop.inria.fr/everest/Marieke.Huisman/mt.pdf
13. Kiefer, S., Schwoon, S., Suwimonteerabuth, D.: http://www.informatik.uni-stuttgart.de/fmi/szs/tools/moped/
14. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
15. Kupferman, O., Vardi, M.: An automata-theoretic approach to modular model checking. ACM TOPLAS 22(1), 87–128 (2000)
16. Lindholm, T., Yellin, F.: The Java^{TM} Virtual Machine Specification, 2nd edn. Sun Microsystems, Inc. (1999)
17. Manson, J., Pugh, W., Adve, S.: The Java memory model. In: Principles of Programming Languages (2005)
18. Méndez, M., Navas, J., Hermenegildo, M.V.: An efficient, parametric fixpoint algorithm for analysis of Java bytecode. In: Huisman, M., Spoto, F. (eds.) Bytecode 2007, pp. 51–66 (2007)
19. Obdržálek, J.: Model checking Java using pushdown systems. In: Proceedings of FTfJP 2002, Malaga, Available as Technical Report NIII-R0204, Computing Science Department, University of Nijmegen (June 2002)
20. Polansky, D.: Implementation of the model checker for pushdown systems and alternation-free mu-calculus. Master's thesis, FI MU Brno (2000)
21. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
22. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM TOPLAS 22(2), 416–430 (2000)
23. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A Java bytecode checker based on Moped. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 541–545. Springer, Heidelberg (2005)
24. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java Optimization Framework. In: CASCON 1999, pp. 125–135 (1999)

A Unified Model Checking Approach with Projection Temporal Logic*

Zhenhua Duan and Cong Tian

Institute of Computing Theory and Technology, Xidian University,
Xi'an, 710071, P.R. China
{zhhduan,ctian}@mail.xidian.edu.cn

Abstract. This paper presents a unified model checking approach with Projection Temporal Logic (PTL) based on Normal Form Graphs (NFGs). To this end, a Modeling, Simulation and Verification Language (MSVL) is defined based on PTL. Further, normal forms and NFGs for MSVL programs and Propositional PTL (PPTL) formulas are defined. The finiteness for NFGs of MSVL programs is proved in details. Moreover, by modeling a system with an MSVL program p, and specifying the desirable property of the system with a PPTL formula ϕ, whether or not the system satisfies the property (whether or not $p \rightarrow \phi$ is valid) can equivalently be checked by evaluating whether or not $\neg(p \rightarrow \phi) \equiv p \wedge \neg\phi$ is unsatisfiable. Finally, the satisfiability of a formula in the form of $p \wedge \neg\phi$ is checked by constructing the NFG of $p \wedge \neg\phi$, and then inspecting whether or not there exist paths in the NFG.

1 Introduction

Verification and testing are basic techniques to validate systems [11,23,24] at the present. Model checking is an automatic verification approach based on model theory. To verify whether or not a system meets a property, the system is modeled as a finite transition system or automaton M, and the property is specified by a temporal logic formula p. Then a model checking procedure is employed to check whether or not $M \models p$. The advantage of model checking is that the verification can be done automatically. However, it suffers from the state explosion problem. Also, it is less suitable for data intensive applications since the treatment of data usually produces infinite state spaces [8]. Two successful model checking tools are SPIN [7] and SMV [8].

The state explosion problem is typically caused by models growing exponentially in the number of parallel components or data elements of an argument system. This observation has led to a number of techniques for fighting this problem. The most rigorous approaches are compositional ones [12,17,18,19], trying to avoid the problem in a divide and conquer fashion. Partial order methods

* This research is supported by the NSFC Grant No. 60433010, and Defense Pre-Research Foundation of China, Grant No. 51315050105.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 167–186, 2008.

limit the size of the models representation by suppressing unnecessary interleavings, which typically arise as a result of the serialization during the model construction of concurrent systems [13,14,15,16].

The most significant improvement to model checking is made by Symbolic Model Checking (SMC) [8,9,10] and Bounded Model Checking (BMC) [20]. In SMC, sets of states are represented implicitly using boolean functions which can be manipulated efficiently with Reduced Ordered Binary Decision Diagram (ROBDD, or BDD for short) [21]. As a result, SMC allows a polynomial system representation but may explode in the course of the model checking process. The combination of SMC with BDDs pushed the barrier to systems with 10^{20} states and more [10]. However, the bottleneck of SMC methods is the amount of memory that is required for storing and manipulating BDDs. Although numerous techniques such as decomposition, abstraction, and various reductions have been proposed through the years to overcome this problem, full verification of many designs is still beyond the capacity of BDD based SMC.

The basic idea in BMC is to search for a counterexample in executions whose length is bounded by some integer k [20]. If no bug is found then we increases k until either a bug is found, the problem becomes intractable, or some pre-known upper bound is reached. The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods rather than BDDs. Experiments have shown that it can solve many systems that cannot be solved by BDD-based techniques. However, BMC does not solve the complexity problem of model checking since it still relies on an exponential procedure and hence is limited in its capacity. BMC also has the disadvantage of not being able to prove the absence of errors.

In this article, we present a unified model checking approach with Projection Temporal Logic (PTL) based on Normal Form Graphs (NFGs) [5]. With this method, a system is first modeled as P using a modeling, simulation and verification language called MSVL which is a subset of PTL [2,25] and an extension of Framed Tempura [6]. Thus, P is a non-deterministic program of MSVL and also a formula of PTL. Second, a property of the system is specified by a formula ϕ of Propositional PTL (PPTL) [2,5]. To check whether or not P satisfies ϕ amounts to proving $\models P \rightarrow \phi$. It turns out equivalently to prove $\not\models P \wedge \neg\phi$. Thus, we translate the model checking problem into a satisfiability problem in PPTL since finite state programs in MSVL are equivalent to PPTL formulas (see Appendix C). As a result, we have proved that PPTL is decidable and given a decision procedure in [5]. With this procedure, a PPTL formula is satisfiable if and only if there is a valid path in its NFG. Therefore, the problem of checking whether or not P satisfies ϕ is eventually translated to the problem of checking whether or not the NFG of $P \wedge \neg\phi$ contains a valid finite or infinite path. If not, the property is verified otherwise a valid path of the NFG determines a counterexample. Based on the above analysis, a model checking algorithm can be given as follows: (1) modeling the system as program P in MSVL and specifying the property of the system as a PPTL formula ϕ; (2) constructing the NFG of $P \wedge \neg\phi$; (3) checking the NFG to find out a counterexample if the NFG contains valid paths

otherwise output 'satisfied' message. However, a further analysis tells us that a more effective recursive algorithm can be given since we can transform P and $\neg\phi$ into their normal forms separately and the conjunction of P and $\neg\phi$ can be reduced to the form in $P_e \wedge \phi_e \wedge \varepsilon \vee \bigcirc(P' \wedge \neg\phi')$. Thus, the NFG of the original formula $P \wedge \neg\phi$ can recursively be constructed.

Our method has some advantages. For instance, (1) the model and property of a system can be written in the same logic; (2) the model checking algorithm is relies on constructing the NFG of a PPTL formula; during the construction, when a valid finite or infinite path has been constructed the algorithm immediately stops since we do not need to construct the whole NFG of the formula if we do not expect to have all counterexamples; (3) the existing SAT procedure can be reused to check the satisfaction of the state formulas with the present components of a normal form; (4) the expressiveness of PPTL is more powerful than Propositional Linear TL (PLTL) since we have proved that the expressiveness of PPTL is equivalent to the full regular expression [22] but that of PLTL equals star free regular expression [26,27]. However, in the worst case, our model checking approach does not solve the complexity problem of model checking since it still relies on an exponential procedure and hence is limited in its capacity.

This paper is organized as follows. In the following section, the syntax, semantics and some logic laws of PTL are presented. In Section 3, the language MSVL is formalized, the normal form and NFG of MSVL are defined, and finiteness for NFGs of MSVL is proved. Correspondingly, as a property specification language, the syntax, semantics, normal form and NFGs of PPTL formulas are briefly introduced in Section 4. In Section 5, the unified model checking approach with PTL based NFGs is presented. Further, an example is given to show how the model checking algorithm and the developed supporting tools work. Finally, conclusions are drawn in Section 6.

2 Projection Temporal Logic

Our underlying logic is Projection Temporal Logic [3,2], it is an extension of Interval Temporal Logic (ITL) [4]. Let Π be a countable set of propositions, and V be a countable set of typed static and dynamic variables. $B = \{true, false\}$ represents the boolean domain and D denotes all the data we need. The terms e and formulas p of the logic are given by the following grammar:

$$e ::= \upsilon \mid \bigcirc e \mid \ominus e \mid beg(e) \mid end(e) \mid f(e_1, ..., e_n)$$

$$p ::= \pi \mid e_1 = e_2 \mid P(e_1, ..., e_n) \mid \neg p \mid p_1 \wedge p_2 \mid \exists x : p \mid \bigcirc p \mid (p_1, ..., p_m) prj\ p \mid p^+$$

where $\pi \in \Pi$ is a proposition, and υ a dynamic variable or a static variable. In $f(e_1, \ldots, e_n)$ and $P(e_1, \ldots, e_n)$, 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. $\bigcirc$, $\ominus$, $beg(.)$, $end(.)$ and prj); otherwise it is a temporal formula (term).

A state s is a pair of assignments (I_v, I_p) where for each variable $v \in V$ defines $s[v] = I_v[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi] = I_p[\pi]$. $I_v[v]$ is a value in D or nil (undefined), whereas $I_p[\pi] \in B$. An interval $\sigma = \langle s_0, s_1, \dots \rangle$ is a non-empty (possibly infinite) sequence of states. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of states in σ minus one. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set N_0 of non-negative integers and ω, $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0$, $i < \omega$. Moreover, we define $\preceq$ as $\leq - \{(\omega, \omega)\}$. With such a notation, $\sigma_{(i..j)}$ $(0 \leq i \leq j \leq |\sigma|)$ denotes the sub-interval $< s_i, ..., s_j >$ and $\sigma^{(k)}$ $(0 \leq k \leq |\sigma|)$ denotes $< s_k, ..., s_{|\sigma|} >$. The concatenation of σ with another interval (or empty string) σ' is denoted by $\sigma \cdot \sigma'$. To define the semantics of the projection operator we need an auxiliary operator for intervals. Let $\sigma = \langle s_0, s_1, \dots \rangle$ be an interval and $r_1, \dots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \dots \leq r_h \leq |\sigma|$. The projection of σ onto $r_1, \dots, r_h$ is the interval (called projected interval), $\sigma \downarrow (r_1, \dots, r_h) = \langle s_{t_1}, s_{t_2}, \dots, s_{t_l} \rangle$, where $t_1, \dots, t_l$ is obtained from $r_1, \dots, r_h$ by deleting all duplicates. For example,

$$\langle s_0, s_1, s_2, s_3, s_4 \rangle \downarrow (0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3 \rangle$$

An interpretation for a PTL term or formula is a tuple $\mathcal{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \dots \rangle$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \leq j \leq |\sigma|$. 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. For every term e, the evaluation of e relative to interpretation $\mathcal{I} = (\sigma, i, k, j)$ is defined as $\mathcal{I}[e]$, by induction on the structure of a term, where v is a variable and $e_1, \dots, e_m$ are terms.

$$\begin{aligned}
\mathcal{I}[v] &= s_k[v] \;=\; I_v^k[v] \;=\; I_v^i[v], \text{ if } v \text{ is a static variable.} \\
\mathcal{I}[v] &= s_k[v] \;=\; I_v^k[v], \text{ if } v \text{ is a dynamic variable.} \\
\mathcal{I}[f(e_1, \dots, e_m)] &= \begin{cases} f(\mathcal{I}[e_1], \dots, \mathcal{I}[e_m]), \text{ if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil, \text{otherwise} \end{cases} \\
\mathcal{I}[\bigcirc e] &= \begin{cases} (\sigma, i, k+1, j)[e], \text{ if } k < j \\ nil, \text{ otherwise} \end{cases} \\
\mathcal{I}[\ominus e] &= \begin{cases} (\sigma, i, k-1, j)[e], \text{ if } i < k \\ nil, \text{ otherwise} \end{cases} \\
\mathcal{I}[beg(e)] &= (\sigma, i, i, j)[e] \\
\mathcal{I}[end(e)] &= \begin{cases} (\sigma, i, j, j)[e], \text{ if } j \neq \omega \\ nil, \text{ otherwise} \end{cases}
\end{aligned}$$

The satisfaction relation for formulas $\models$ is inductively defined as follows.

1. $\mathcal{I} \models \pi$ if $s_k[\pi] = I_p^k[\pi] = true$.
2. $\mathcal{I} \models P(e_1, \dots, e_m)$ if $P(\mathcal{I}[e_1], \dots, \mathcal{I}[e_m]) = true$ and $\mathcal{I}[e_h] \neq nil$, for all h.
3. $\mathcal{I} \models e = e'$ if $\mathcal{I}[e] = \mathcal{I}[e']$.
4. $\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.
5. $\mathcal{I} \models p \wedge q$ if $\mathcal{I} \models p$ and $\mathcal{I} \models q$.

6. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.
7. $\mathcal{I} \models \exists x : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be in the values assigned to variable x.
8. $\mathcal{I} \models (p_1, \ldots, p_m)\ prj\ q$ if there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \preceq j$ such that $(\sigma, i, r_0, r_1) \models p_1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$
   (b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.
9. $\mathcal{I} \models p^+$ if there are $k = r_0 \leq r_1 \leq ... \leq r_{n-1} \preceq r_n = j$ $(n \geq 1)$ such that $(\sigma, i, r_0, r_1) \models p$ and $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p$ for all $1 < l \leq n$.

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 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 lec-formula if $(\sigma, k, k, j) \models p \Leftrightarrow (\sigma, i, k, j) \models p$ for any interpretation (σ, i, k, j).

The abbreviations $true$, $false$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $true \stackrel{\text{def}}{=} P \vee \neg P$ and $false \stackrel{\text{def}}{=} P \wedge \neg P$ for any formula P. Also we have the following derived formulas:

$$
\begin{array}{rcl@{\qquad}rcl}
empty & \stackrel{\text{def}}{=} & \neg \bigcirc true & more & \stackrel{\text{def}}{=} & \neg empty \\
len(0) & \stackrel{\text{def}}{=} & empty & len(n) & \stackrel{\text{def}}{=} & \bigcirc len(n-1), n \geq 1 \\
skip & \stackrel{\text{def}}{=} & len(1) & \odot P & \stackrel{\text{def}}{=} & empty \vee \bigcirc P \\
P;\ Q & \stackrel{\text{def}}{=} & (P,\ Q)\ prj\ empty & \Diamond P & \stackrel{\text{def}}{=} & true\ ;\ P \\
\Box P & \stackrel{\text{def}}{=} & \neg \Diamond \neg P & p^* & \stackrel{\text{def}}{=} & empty \vee p^+
\end{array}
$$

Some useful logic laws of PTL can be found in Appendix A and their proofs can be found in [5,6].

3 Modeling, Simulation and Verification Language

The Language MSVL is a subset of Projection Temporal Logic with framing technique, and an extension of Framed Tempura [6]. It can be be used for the purpose of modeling, simulation and verification of software and hardware systems.

3.1 Framing

Framing is concerned with the persistence of the values of variables from one state to another. Intuitively, the framing operation on variable x, denoted by $frame(x)$, means that variable x always keeps its old value over an interval if no assignment to x is encountered. For the definition of $frame$ operator, a new assignment called *a positive immediate assignment* is defined as

$$x \Leftarrow e \stackrel{\text{def}}{=} x = e \wedge p_x$$

where p_x is an atomic proposition associated with state (dynamic) variable x, and notice that p_x cannot be used for other purpose. To identify an occurrence of an assignment to a variable, say x, we make use of a flag called the assignment flag, denoted by a predicate $af(x)$; it is true whenever an assignment of a value to x is encountered, and false otherwise. The definition of the assignment flag is $af(x) \stackrel{\text{def}}{=} p_x$, for every variable x. There are state framing (lbf) and interval framing ($frame$) operators. Intuitively, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval.

$$\begin{aligned} lbf(x) &\stackrel{\text{def}}{=} \neg af(x) \rightarrow \exists b : (\ominus x = b \wedge x = b) \\ frame(x) &\stackrel{\text{def}}{=} \Box(more \rightarrow \bigcirc lbf(x)) \end{aligned}$$

where b is a static variable.

3.2 The MSVL Language

The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 \ op \ e_1 (op ::= + \mid - \mid * \mid \setminus \mid mod)$$

$$b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows.

Termination:	$empty$
Assignment:	$x = e$
P-I-Assignment:	$x \Leftarrow e$
State Frame:	$lbf(x)$
Interval Frame:	$frame(x)$
Conjunction:	$p \wedge q$
Selection:	$p \vee q$
Next:	$\bigcirc p$
Always:	$\Box p$
Conditional:	$if \ b \ then \ p \ else \ q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
Exists:	$\exists x : p$
Projection:	$(p_1, \ldots, p_m) \ prj \ p$
Sequence:	$p \ ; q$
While:	$while \ b \ do \ p \stackrel{\text{def}}{=} (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$
Parallel:	$p \parallel q \stackrel{\text{def}}{=} (p \wedge (q; true)) \vee (q \wedge (p; true))$
Await:	$await(b) \stackrel{\text{def}}{=} (frame(x_1) \wedge ... \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$ where $x_i \in V_b = \{x \mid x \text{ appears in } b\}$

where x denotes a variable, e stands for an arbitrary arithmetic expression, b a boolean expression, and $p_1, \ldots, p_m$, p and q stand for programs of MSVL. The

assignment $x = e$, positive immediate assignment $x \Leftarrow e$, $empty$, $lbf(x)$, and $frame(x)$ are basic statements and the others are composite ones.

The assignment $x = e$ means that the value of variable x is equal to the value of expression e. Positive immediate assignment $x \Leftarrow e$ indicates that the value of x is equal to the value of e and the assignment flag for variable x, p_x, is $true$. Statements of $if\ b\ then\ p\ else\ q$ and $while\ b\ do\ p$ are the same as that in the conventional imperative languages. The next statement $\bigcirc p$ means that p holds at the next state while $\Box p$ means that p holds at all the states over the whole interval from now. $p \wedge q$ means that p and q are executed concurrently and share all the variables during the mutual execution. $p \vee q$ means p or q are executed. $empty$ is the termination statement meaning that the current state is the final state of the interval over which the program is executed. The sequence statement $p; q$ means that p is executed from the current state to its termination while q will hold at the final state of p and be executed from that state. The existential quantification $\exists x : p$ intends to hide the variable x within the process p. $lbx(x)$ means the value of x in the current state equals to value of x in the previous state if no assignment to x occurs, while $frame(x)$ indicates that the value of variable x always keeps its old value over an interval if no assignment to x is encountered. Different from the conjunction statement, the parallel statement allows both the processes to specify their own intervals. e.g., $len(2) \| len(3)$ holds but $len(2) \wedge len(3)$ is obviously false. Projection can be thought of as a special parallel computation which is executed on different time scales. The projection $(p_1, \ldots, p_m)\ prj\ q$ means that q is executed in parallel with $p_1, \ldots, p_m$ over an interval obtained by taking the endpoints of the intervals over which the $p_i's$ are executed. In particular, the sequence of $p_i's$ and q may terminate at different time points. Finally, $await\ b$ does not change any variable, but waits until the condition b becomes $true$, at which point it terminates.

Further, the following derived statements are useful in practice.

Multiple Selection:	$OR_{k=1}^{n} \stackrel{\text{def}}{=} p_1 \vee p_2 \vee \ldots \vee p_n$
Conditional:	$if\ b\ do\ p \stackrel{\text{def}}{=} if\ b\ do\ p\ else\ empty$
When:	$when\ b\ do\ p \stackrel{\text{def}}{=} await(b); p$
Guarded Command:	$b_1 \rightarrow p_1 \Box \ldots \Box b_n \rightarrow p_n \stackrel{\text{def}}{=} OR_{k=1}^{n}(when\ b_k\ do\ p_k)$
Repeat:	$repeat\ p\ until\ c \stackrel{\text{def}}{=} p; while\ \neg c\ do\ p$

3.3 Normal Forms and NFGs of MSVL

Definition 1. A program q in MSVL is in normal form if

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$$

where $0 \leq l \leq 1$, $t > 0$, and $l + t \geq 1$. For $1 \leq j \leq t$, q_{fj} is a general MSVS program; whereas q_{ei} $(i = 1)$ and q_{cj} $(1 \leq j \leq t)$ are $true$ or all are state formulas of the form:

$$(x_1 = e_1) \wedge \ldots \wedge (x_l = e_l) \wedge \dot{p_{x_1}} \wedge \ldots \wedge \dot{p_{x_l}}$$

where $e_k \in D(1 \leq k \leq l)$. □

Theorem 1. Any MSVL program q can be rewritten into its normal form.

Proof: The proof for transforming most of the statements in MSVL into normal form can be found in [2,6]. The other statements of MSVL can be transformed in a similar way. □

Modeling a system with an MSVL program (formula in PTL) p, according to the normal form, we can construct a graph, namely normal form graph (NFG), which explicitly illustrates the state space of the system. Actually, the NFG also presents the models satisfying formula p [5]. For an MSVL program p, the NFG of p is a directed graph, $G = (CL(p), EL(p))$, where $CL(p)$ denotes the set of nodes and $EL(p)$ denotes the set of edges in the graph. In $CL(p)$, each node is specified by a program in MSVL, while in $EL(p)$, each edge is a directed arc labeled with a state formula p_e from node q to node r and identified by a triple, (q, p_e, r). $CL(p)$ and $EL(p)$ of G can be inductively defined as in Definition 2.

Definition 2. For a program p, the set $CL(p)$ of nodes and the set $EL(p)$ of edges connecting nodes in $CL(p)$ are inductively defined as follows:

1. $p \in CL(p)$;
2. For all $q \in CL(p) \setminus \{\varepsilon, false\}$, if $q \equiv \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$, then $\varepsilon \in CL(p)$, $(q, q_{ei}, \varepsilon) \in EL(p)$ for each i; $q_{fj} \in CL(p)$, $(q, q_{cj}, q_{fj}) \in EL(p)$ for all j;

The NFG of formula p is the directed graph $G = (CL(p), EL(p))$. □

Definition 2 implies an algorithm for constructing NFGs of MSVL programs. In the NFG of a program p generated by Definition 2, the set $CL(p)$ of nodes and the set $EL(p)$ of edges are inductively produced by repeatedly rewriting the new created nodes into their normal forms. So one question we have to answer is whether or not the rewriting process terminates. Fortunately, we can prove that, for any MSVL program p, the number of nodes in $CL(p)$ is finite. An outline of the proof is given in Appendix C.

To precisely characterize the models satisfying the program (formula), that is, the behaviors of the system, a finite label F needs further to be added in the NFG as analyzed in [5].

Example 1. NFG of MSVL program $frame(x) \wedge (x = 2 \vee x = 3) \wedge if\ (x = 2)\ then\ len(2)\ else\ \{len(3)\}$ can be constructed as shown in Fig.1. □

4 Property Specification Language

Propositional PTL (PPTL) is employed as the property specification language in our model checking approach.

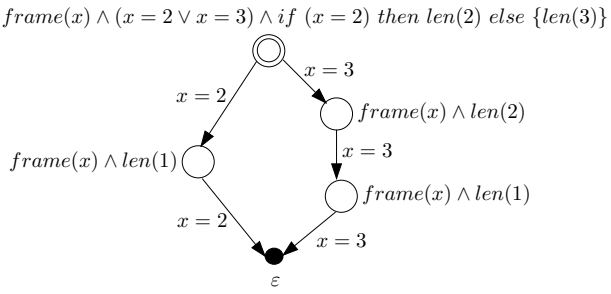


Fig. 1. NFG of MSVL program $frame(x) \wedge (x = 2 \vee x = 3) \wedge if\ (x = 2)\ then\ len(2)\ else\ \{len(3)\}$

4.1 Propositional Projection Temporal Logic

Let *Prop* be a countable set of atomic propositions. The formula p of PPTL is given by the following grammar:

$$p\ ::=\ \pi \mid \bigcirc p \mid \neg p \mid p_1 \vee p_2 \mid (p_1\ ,...,\ p_m)\ prj\ p \mid p^+$$

where $\pi \in Prop$, $p_1\ ,...,\ p_m$ are all well-formed PPTL formulas. A formula is called a state formula if it contains no temporal operators.

Following the definition of Kripke structure [1], we define a state s over *Prop* to be a mapping from *Prop* to $B = \{true, false\}$, $s : Prop \longrightarrow B$. We will use $s[\pi]$ to denote the valuation of π at state s. Intervals, interpretation and satisfaction relation can be defined in the same way as in the first order case.

4.2 Normal Form and NFGs of PPTL Formulas

Definition 3. A PPTL formula q is in normal form if

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$$

where $0 \leq l \leq 1$, $t > 0$, and $l + t \geq 1$, q_{ei} $(i = 1)$ and q_{cj} $(1 \leq j \leq t)$ are *true* or state formulas of the form:

$$\dot{\pi_1} \wedge ... \wedge \dot{\pi_m}$$

where each $\dot{\pi_k} \in Prop$ $(1 \leq k \leq m)$ and $\dot{\pi_k}$ denotes π_k or $\neg\pi_k$. Each q_{fj} is a general PPTL formula. □

Definition 4. In a normal form, if $\bigvee_{j=1}^{t} q_{cj} \equiv true$ and $\bigvee_{i \neq j} (q_{ci} \wedge q_{cj}) \equiv false$, then this normal form is called a complete normal form. □

The complete normal form plays an important role in transforming the negation of a PPTL formula into its normal form. For example, if q has been written to its complete normal form:

$$q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$$

then we have,

$$\neg q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} \neg q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc \neg q_{fj}$$

Theorem 2. Any PPTL formula q can be rewritten into its normal form.
Proof: The proof can be found in [5]. □

A property of a system can be specified by a PPTL formula p. According to the normal form, we can also construct the NFG of p, which explicitly illustrates the models of the formula. The definition for NFGs of PPTL formulas is the same as one defined for MSVL programs.

Theorem 3. For any PPTL formula p, $CL(p)$, the set of nodes in the NFG of p is finite.
Proof: The proof of the theorem can be found in [5]. □

To precisely characterize the models of PPTL formulas, finite labels F are added in the NFGs to confine the finitely often occurrences of some nodes in paths of an NFG as analyzed in [5].

Example 2. NFG of formula $\neg(true; \neg \bigcirc q) \vee p \wedge \bigcirc q$ is shown in Fig.2. □

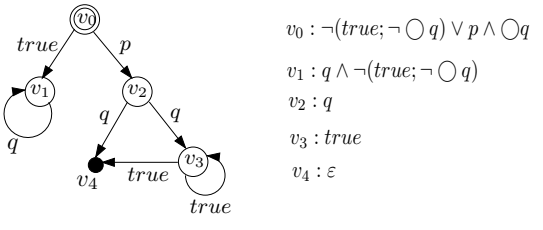


Fig. 2. NFG of formula $\neg(true; \neg \bigcirc q) \vee p \wedge \bigcirc q$

5 Model Checking Approach with PTL Based on NFGs

5.1 Basic Approach

Modeling the system to be verified by an MSVL program p, and specifying the desirable property of the system by a PPTL formula ϕ, to check whether or not the system satisfies the property, we need to prove the validation of

$$p \rightarrow \phi$$

If $p \rightarrow \phi$ valid, the system satisfies the property, otherwise the system violates the property. Equivalently, we can check the satisfiability of

$$\neg(p \rightarrow \phi) \equiv p \wedge \neg\phi$$

If $p \wedge \neg\phi$ is unsatisfiable ($p \rightarrow \phi$ is valid), the system satisfies the property, otherwise the system fails to satisfy the property, and for each $\sigma \models p \wedge \neg\phi$, σ determines a counterexample that the system violates the property. Accordingly, our model checking approach can be translated to the satisfiability of PTL formulas of the form $p \wedge \neg\phi$, where p is an MSVL program and ϕ is a formula in PPTL. Since both model p and property ϕ are formulas in PTL, we call this model checking a unified approach.

To check the satisfiability of PTL formula $p \wedge \neg\phi$, we construct the NFG of $p \wedge \neg\phi$. As depicted in Fig.3, initially, we create the root node $p \wedge \neg\phi$, then

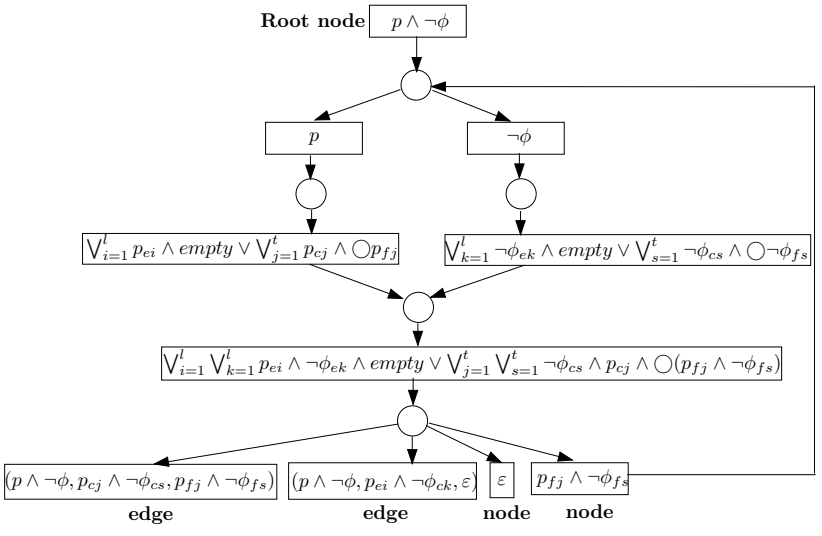


Fig. 3. Constructing NFG of $p \wedge \neg\phi$

we rewrite p and $\neg\phi$ into their normal forms respectively. By computing the conjunction of normal forms of p and $\neg\phi$, new nodes ε and $p_{fj} \wedge \neg\phi_{fs}$, and edges $(p \wedge \neg\phi, p_{ei} \wedge \neg\phi_{ck}, \varepsilon)$ from node $p \wedge \neg\phi$ to ε, $(p \wedge \neg\phi, p_{cj} \wedge \neg\phi_{cs}, p_{fj} \wedge \neg\phi_{fs})$ from $p \wedge \neg\phi$ to $p_{fj} \wedge \neg\phi_{fs}$ are created. Further, by dealing with each new created nodes $p_{fj} \wedge \neg\phi_{fs}$ using the same methods as the root nodes $p \wedge \neg\phi$ repeatedly, the NFG of $p \wedge \neg\phi$ can be produced. Thus, it is apparent that each node in the NFG of $p \wedge \neg\phi$ is in the form of $p' \wedge \neg\phi'$, where p' and ϕ' are nodes in the NFGs of p and $\neg\phi$ respectively. Therefore, a recursive algorithm can be formalized in Pseudo code as shown in algorithm *NFG*. In the algorithm, another function $Nf(p)$ is called to produce the normal form of a PPTL formula or an MSVL program p. This function can be found in [5]. For the complexity of the algorithm, roughly speaking, if $|cl(p)| = O(n)$ and $|cl(\neg\phi)| = O(m)$, at most, $|cl(p \wedge \neg\phi)| = O(n \times m)$.

Function NFG($p \wedge \neg\phi$)
/* precondition: p is a program in MSVL, $\neg\phi$ is a formula in PPTL*/
/* postcondition: NFG($p \wedge \neg\phi$) computes NFG of $p \wedge \neg\phi$, $G = (CL(p \wedge \neg\phi), EL(p \wedge \neg\phi))$*/
begin function
$CL(p \wedge \neg\phi) = \{p \wedge \neg\phi\}$; $EL(p \wedge \neg\phi) = \emptyset$; $mark[p \wedge \neg\phi] = 0$; AddE = AddN = 0;
while there exists $r \wedge \neg\varphi \in CL(p \wedge \neg\phi) \setminus \{\varepsilon\}$, and $mark[r \wedge \neg\varphi]$ ==0
do $mark[r \wedge \neg\varphi]$ =1; /*marking $r \wedge \neg\varphi$ is decomposed*/
Q =NF(r)∧NF($\neg\varphi$);
case
Q is $\bigvee_{j=1}^{h} \bigvee_{i=1}^{t} r_{ej} \wedge \neg\varphi_{ei} \wedge empty$: AddE=1; /*first part of NF needs added*/
Q is $\bigvee_{k=1}^{t} \bigvee_{l=1}^{n} r_{ck} \wedge \neg\varphi_{cl} \wedge \bigcirc(r_{fk} \wedge \neg\varphi_{fl})$: AddN=1; /*second part of NF needs added*/
Q is $\bigvee_{j=1}^{h} \bigvee_{i=1}^{t} r_{ej} \wedge \neg\varphi_{ei} \wedge empty \vee \bigvee_{k=1}^{t} \bigvee_{l=1}^{n} r_{ck} \wedge \neg\varphi_{cl} \wedge \bigcirc(r_{fk} \wedge \neg\varphi_{fl})$: AddE=AddN=1;
/*both parts of NF needs added*/
end case
if AddE == 1 **then** /*add first part of NF*/
$CL(p \wedge \neg\phi) = CL(p \wedge \neg\phi) \cup \{\varepsilon\}$;
$EL(p \wedge \neg\phi) = EL(p \wedge \neg\phi) \cup \bigcup_{j=1}^{h} \bigcup_{i=1}^{t} \{(r \wedge \neg\varphi, r_{ej} \wedge \neg\varphi_{ei}, \varepsilon)\}$;
AddE=0;
if AddN == 1 **then** /*add second part of NF*/
for each $r_{fk} \wedge \neg\varphi_{fl}$ **if** $r_{fk} \wedge \neg\varphi_{fl} \notin CL(p \wedge \neg\phi)$
$mark[r_{fk} \wedge \neg\varphi_{fl}]$=0; /*$r_{fk} \wedge \neg\varphi_{fl}$ needs decomposed*/
$CL(p \wedge \neg\phi) = CL(p \wedge \neg\phi) \cup \bigcup_{k=1}^{t} \bigcup_{l=1}^{n} \{r_{fk} \wedge \neg\varphi_{fl}\}$;
$EL(p \wedge \neg\phi) = EL(p \wedge \neg\phi) \cup \bigcup_{k=1}^{t} \bigcup_{l=1}^{n} \{(r \wedge \neg\varphi, r_{ck} \wedge \neg\varphi_{cl}, r_{fk} \wedge \neg\varphi_{fl})\}$;
AddN=0;
end while
return G;
end function

Further, for any node in the NFG of $p \wedge \neg\phi$, finite label F is added in node $p' \wedge \neg\phi'$ where if in the NFG of p, p' is labeled with F, or in the NFG of $\neg\phi$, $\neg\phi'$ is labeled with F. In the NFG of formula $q \equiv p \wedge \neg\phi$, a finite path, $\Pi = \langle q, q_e, q_1, q_{1e}, ..., \varepsilon\rangle$, is an alternate sequence of nodes and edges from the root to ε node, while an infinite path, $\Pi = \langle q, q_e, q_1, q_{1e}, ...\rangle$, is an infinite alternate sequence of nodes and edges emanating from the root, where F occurs only finitely often. Similar to the proof in [5], it can be proved that, the paths in the NFG of q precisely characterize models of q. Thus, if there exist paths in the NFG of q, q is satisfiable, otherwise unsatisfiable.

5.2 Model Checker

We have developed a model checking tool (prototype) based on our model checking algorithm. Generally, the prototype can work in three modes: modeling, simulation and verification. With the modeling mode, given the MSVL program p of a system, the state space of the system can implicitly be given as an NFG of p. In the simulation mode, an execution path of the NFG of the system is output according to minimal model semantics of MSVL [6]. Under the verification mode, given a system model described by an MSVL program, and a property specified by a PPTL formula, it can automatically be checked whether the system

satisfies the property or not, and the counterexample can be given if the system does not satisfy the property.

5.3 Example

As an example, consider the mutual exclusion problem of two processes competing for a shared resource as analyzed in [20]. Pseudo code for this example can be given as shown in Fig.4. We assume that the processes are executed in one time unit in

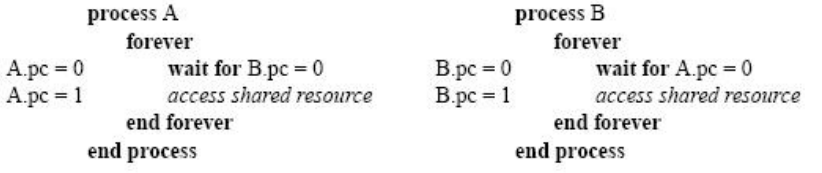


Fig. 4. Pseudo code for two processes A and B competing for a shared resource

an interleaving manner. The wait statement makes a process into sleep. When all processes are asleep the scheduler tries to find a process satisfying waiting condition and reactivates the corresponding process. If all of the waiting conditions are false the system stalls. This mutual exclusion problem can be coded in MSVL as follows. Notice that the underlined code can be ignored with the current part since it is for the purpose of making a counterexample later on.

frame(Apc, Bpc, Ars, Brs) and
(Apc=0 and Ars=0 and Bpc=0 and Brs=0 and skip;
while(true){
(await(Bpc=0);
Apc=1 and Ars=1 and skip;
Apc=0 and Ars=0 and skip)
or
(Apc=1 and Ars=1 and Bpc=1 and Brs=1 and skip;
Apc=0 and Ars=0 and Bpc=0 and Brs=0 and skip)
or
(await(Apc=0);
Bpc=1 and Brs=1 and skip;
Bpc=0 and Brs=0 and skip) }).

where *Ars=1* (*Brs=1*) means processes A (B) is in the shared resource, while *Ars=0* (*Brs=0*) means processes A (B) has released the shared resource. With the modeling mode of MSVL, the state space of the mutual exclusion problem can be created and presented as an NFG as shown in Fig.5. In the NFG, edge 0 indicates that neither process A nor B is in the shared resource; edge 1 (from 1 to 2) indicates that process A is in the shared resource and B is not; edge 2 (from 2 to 1) indicates that neither process A nor B is in the shared resource; edge 4 (from 1 to 3) indicates that process B is in the shared resource and A is not; edge 5 (from 3 to 1) indicates that neither process A nor B is in the shared resource.

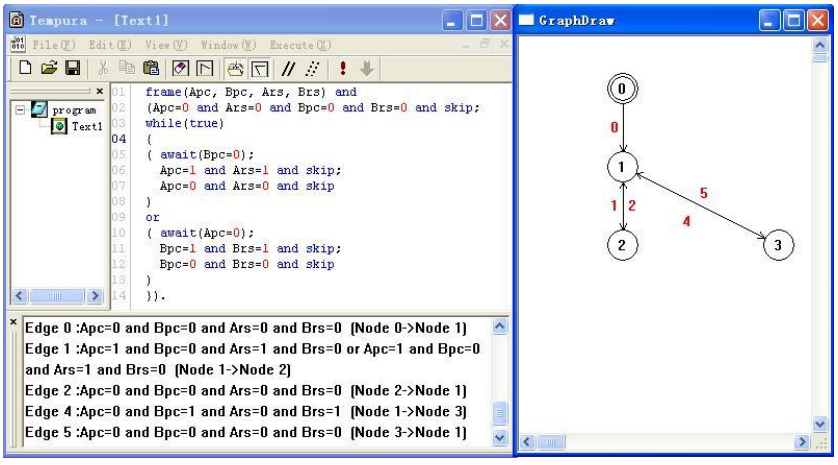


Fig. 5. NFG of the mutual exclusion problem

As a result, the property, "processes A and B will never be in the shared resource in the same time", should hold. That is, Ars and Brs will never be assigned with 1 at the same time. By employing propositions p and q to denote $Ars = 1$ and $Brs = 1$ respectively, this property can be specified by $\Box(\neg(p \wedge q))$ in PPTL. With the verification mode of MSVL, we add the following code

</define p:Ars=1; define q:Brs=1; always(∼(p and q))/>

to the beginning of the MSVL code for the mutual exclusion problem, then run the code with model checker, an empty NFG with no edges is produced as shown in Fig.6. This means that the formula is unsatisfiable, and the system satisfies the property.

Fig. 6. Verification result

Suppose that, when A is in the shared resource, B is possible in the shared resource. To model it, we add the code with underline to the previous MSVL code of the system. Now we check whether or not the system satisfies $\Box(\neg(p \wedge q))$, and the resulting NFG is produced as shown in Fig.7. Obviously, there exist infinite paths where node 1 and node 2 appear infinitely often. Thus, the property cannot be satisfied.

As you can see, MSVL and PPTL can be used to verify properties of programs in a similar way as kripke structures (or automata) and PLTL (or CTL) do. However, the expressive power of PLTL and CTL is limited. They cannot express regular properties such as "a property Q holds at even states ignoring odd ones

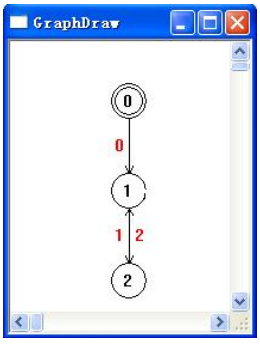


Fig. 7. Verification result

over an interval (or computation run) " [7]. This type of property can be specified and verified by PPTL. In the following, we further verify such a property of the mutual exclusion problem.

The mutual exclusion problem has a special property: "neither A nor B is in the shared resource, immediately after A or B released the shared resource; and when A or B is in the shared resource, it releases the resource at the next state". Basically, this property is a regular property. It can be specified by, $\neg p \wedge \neg q \wedge (\bigcirc^2 \neg p \wedge \neg q \wedge empty)^*$ in PPTL. Thus, we can add *</define p:Ars=1; define q:Brs=1; ∼p and ∼q and (next(next(∼p and ∼q and empty)))# />* to the beginning of the MSVL code for the mutual exclusion problem, then run the code with model checker, an empty NFG with no edges is produced as shown in Fig.8. Hence, the formula is unsatisfiable, and the system satisfies the property.

Fig. 8. Verification result

6 Conclusion

In this paper, we proposed a unified model checking approach with PTL based on NFGs. A model checker has also been developed to support the proposed method. This approach has an apparent advantage: model p and property ϕ of a system are both described in the same logic framework PTL. This enables us to translate the problem of checking whether or not p satisfies ϕ to the problem of checking the satisfiability of $p \wedge \neg\phi$. In turn, this can be done by constructing the NFG of $p \wedge \neg\phi$ and checking whether or not there exist any finite or infinite paths in the NFG. As you can see, NFG is a finite graph based structure. So we can use graph theory to manipulate NFGs. Further, an NFG can be equivalently transformed to a Büchi automaton [22]. Hence, automata theory can also be used to manipulate NFGs. However, our approach, in worst case, does not reduce the

complexity of the model checking problems although in many cases it works well since we do not need to produce a whole NFG but just a finite or infinite path as a counterexample.

To combat the space explosion problem, we will further investigate the possibility of combinations of SMC or BMC techniques with our approach. In particular, BMC is a SAT based approach for searching a counterexample in a given integer k steps. With this approach, the model M in the Kripke structure and property ϕ in a PLTL formula of a system are translated to a propositional classic logic formula f. To check whether or not $M \models \phi$ is equivalently to check the satisfiability of f. Thus, the SAT procedure can be used to solve the problem. This idea can be used in our approach. However, we do not need to translate the formulas into a classic propositional logic framework rather in their normal forms and use SAT procedures in a stepwise way. This research is a challenge to us in the near future. Also, the current version of the model checker is merely a prototype and lots of efforts are needed to improve it. In addition, to examine our method, several case studies with larger examples are also required.

Acknowledgement

We would like to thank Miss Xiao Xiao Yang, Miss Xia Guo and Miss Xiao Xing Zhang for the useful discussion. In particular, Guo and Zhang's effort to make the verification example work with the prototype is very appreciated.

References

1. Kripke, S.A.: Semantical analysis of modal logic I: normal propositional calculi. Z. Math. Logik Grund. Math. 9, 67–96 (1963)
2. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne (May 1996)
3. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)
4. Moszkowski, B.: Reasoning about digital circuits. Ph.D Thesis, Department of Computer Science, Stanford University. TRSTAN-CS-83-970 (1983)
5. Duan, Z., Tian, C., Zhang, L.: A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. Acta Informatica 45(1), 43–78 (2008)
6. Duan, Z., Yang, X., Koutny, M.: Framed Temporal Logic Programming. Science of Computer Programming 70, 31–61 (2008)
7. Holzmann, G.J.: The Model Checker Spin. IEEE Trans. on Software Engineering 23(5), 279–295 (1997)
8. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Information and Computation 98(2), 142–170 (1992)
9. Coudert, O., Madre, J.C.: A unified framework for the formal verification of sequential circuits. In: Proc. IEEE International Conference on Computer-Aided Design (1990)

10. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579. Springer, Heidelberg (1999)
11. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
12. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. ASI, vol. F 13, pp. 123–144. Springer, Berlin (1985)
13. Valmari, A.: A stubborn attack on state explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
14. Godefroid, P., Wolper, P.: A partial approach to model checking. Information and Computation 110(2), 305–326 (1994)
15. Esparza, J.: Model checking using net unfoldings. Science of Computer Programming 23, 151–195 (1994)
16. Penczek, W., Gerth, R., Kuiper, R.: Partial order reductions preserving simulations (submitted for publication, 1999)
17. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
18. Josko, B.: Verifying the correctness of AADL modules using model checking. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) REX 1989. LNCS, vol. 430, pp. 386–400. Springer, Heidelberg (1990)
19. Josko, B.: Modular Specification and Verification of Reactive Systems. PhD thesis, Univ. Oldenburg, Fachbereich Informatik (April 1993)
20. Biere, A., Cimati, A., Clark, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers 58 (2003)
21. Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C35(12), 1035–1044 (1986)
22. Tian, C., Duan, Z.: Propositional Projection Temporal Logic. In: Agrawal, M., Du, D., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
23. Liu, S., Wang, H.: An automated approach to specification animation for validation. Journal of Systems and Software 80, 1271–1285 (2007)
24. Liu, S., Chen, Y.: A relation-based method combining functional and structural testing for test case generation. Journal of Systems and Software 81, 234–248 (2008)
25. Duan, Z., Koutny, M.: A framed temporal logic programming language. Journal of Computer Science and Technology 19, 333–344 (2004)
26. Gabbay, D., Pnueli, A., Shelah, S., Stavi, J.: On the temporal analysis of fairness. In: POPL 1980: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 163–173. ACM Press, New York (1980)
27. McNaughton, R., Papert, S.A.: Counter-Free Automata (M.I.T research monograph no.65). The MIT Press, Cambridge (1971)

Appendix A: Logic Laws of PTL

The following are some useful logic laws of PTL.

L_1 $\Box(P \wedge Q) \equiv \Box P \wedge \Box Q$

L_2 $\Diamond(P \vee Q) \equiv \Diamond P \vee \Diamond Q$

L_3 $\bigcirc(P \vee Q) \equiv \bigcirc P \vee \bigcirc Q$

L_4 $\bigcirc(P \wedge Q) \equiv \bigcirc P \wedge \bigcirc Q$

L_5 $R;(P \vee Q) \equiv (R;P) \vee (R;Q)$

L_6 $(P \vee Q);R \equiv (P;R) \vee (Q;R)$

L_7 $\Diamond P \equiv P \vee \bigcirc \Diamond P$

L_8 $\Box P \equiv P \wedge \odot \Box P$

L_9 $more \wedge \neg \bigcirc P \equiv more \wedge \bigcirc \neg P$

L_{10} $\neg \odot P \equiv \bigcirc \neg P$

L_{11} $\bigcirc(\exists x : p) \equiv \exists x : \bigcirc p$

L_{12} $\bigcirc P; Q \equiv \bigcirc(P\ ; Q)$

L_{13} $w \wedge (P\ ; Q) \equiv (w \wedge P)\ ; Q$

L_{14} $p^+ \equiv p \vee (p; p^+)$

L_{15} $Q\ prj\ empty \equiv Q$

L_{16} $empty\ prj\ Q \equiv Q$

L_{17} $(P_1, ...P_m)\ prj\ empty \equiv P_1; ...; P_m$

L_{18} $(P, empty)\ prj\ Q \equiv (P \wedge \Diamond empty)\ prj\ Q$

L_{19} $(P_1, ..., P_t, w \wedge empty, P_{t+1}, ..., P_m)\ prj\ Q \equiv (P_1, ..., P_t, w \wedge P_{t+1}, ..., P_m) prj Q$

L_{20} $(P_1, ..., (P_i \vee P_i'), ..., P_m)\ prj\ Q \equiv ((P_1, ..., P_i, ..., P_m)\ prj\ Q) \vee ((P_1, ..., P_i', ..., P_m)\ prj\ Q)$

L_{21} $(P_1, ..., P_m)\ prj\ (P \vee Q) \equiv ((P_1, ..., P_m)\ prj\ P) \vee ((P_1, ..., P_m)\ prj\ Q)$

L_{22} $(P_1, ..., P_m)\ prj\ \bigcirc Q \equiv (P_1 \wedge more; (P_2, ..., P_m)\ prj\ Q) \vee (P_1 \wedge empty; (P_2, ..., P_m)\ prj\ \bigcirc Q)$

L_{23} $(\bigcirc P_1, ..., P_m)\ prj\ \bigcirc Q \equiv \bigcirc(P_1\ ; (P_2, ..., P_m)\ prj\ Q)$

L_{24} $(w \wedge P_1, ..., P_m)\ prj\ Q \equiv w \wedge ((P_1, ..., P_m)\ prj\ Q)$

L_{25} $(P_1, ..., P_m)\ prj\ (w \wedge Q) \equiv w \wedge ((P_1, ..., P_m)\ prj\ Q)$

Appendix B: Logic Laws of MSVL

L_{26}	$while\ b\ do\ p \equiv if\ b\ then\ (p; while\ b\ do\ p)\ else\ empty$
L_{27}	$while\ b\ do\ p \equiv if\ b\ then\ (p \wedge more; while\ b\ do\ p)\ else\ empty$
L_{28}	$while\ b\ do\ p \equiv ((\neg b \wedge empty) \vee (b \wedge p \wedge more; while\ b\ do\ p)) \vee b \wedge p \wedge \Box more$
L_{29}	$while\ b\ do\ p \equiv ((\neg b \wedge empty) \vee (b \wedge p; while\ b\ do\ p)) \vee b \wedge p \wedge \Box more$
L_{30}	$frame(x) \equiv frame(x)\|\|frame(x) \equiv frame(x); frame(x) \equiv frame(x) \wedge frame(x)$
L_{31}	$frame(x) \wedge more \equiv \bigcirc(lbf(x) \wedge frame(x))$
L_{32}	$frame(x) \wedge empty \equiv empty$
L_{33}	$frame(x) \wedge (p \vee q) \equiv frame(x) \wedge p \vee frame(x) \wedge q$
L_{34}	$frame(x) \wedge (p; q) \equiv frame(x) \wedge p; frame(x) \wedge q$
L_{35}	$frame(x) \wedge (p\|\|q) \equiv frame(x) \wedge p\|\|frame(x) \wedge q$

Appendix C: Finiteness of NFGs of MSVL Programs

Let $D = \{d_1, ..., d_n\}$ be a finite set of data, $V = \{x_1, ..., x_m\}$ a finite set of variables, and $Prop$ a countable set of atomic propositions. To prove the finiteness of NFGs of MSVL programs, we first prove that, for any MSVL program, it can be equivalently expressed by a PPTL formula.

Theorem 4. Any program p in MSVL can be equivalently expressed by a formula $\Phi(p)$ in PPTL.

Proof. The proof proceeds by induction on structures of programs in MSVL. First of all, we assume that any boolean expression b can be evaluated to a boolean value true or false, and an arithmetic expression e can be evaluated to a value $d_k \in D$. Therefore, a boolean expression b can be thought of as an atomic proposition $p_b \in Prop$. Further,

1. For $empty$, $\Phi(empty) \stackrel{\text{def}}{=} empty$;
2. For $x_i = e$, we define $x_i = d_j \stackrel{\text{def}}{=} p_i^j \in Prop$, where $x_i \in V$ and $d_j \in D$. Thus, $\Phi(x_i = e) \stackrel{\text{def}}{=} p_i^k$ if $e = d_k \in D$ otherwise $false$
3. For $x_i \Leftarrow e$, by the definition, $x_i \Leftarrow e \equiv x_i = e \wedge p_{x_i}$,

$$\Phi(x_i \Leftarrow e) \stackrel{\text{def}}{=} \Phi(x_i = e) \wedge p_{x_i}$$

4. For $lbf(x_i)$, by the definition, $lbf(x_i) \equiv \neg p_{x_i} \rightarrow \exists d_n \in D : (\ominus x_i = d_n \wedge x_i = d_n)$, we have,

$$\Phi(lbf(x_i)) \stackrel{\text{def}}{=} \neg p_{x_i} \rightarrow p_i^n$$

5. For $frame(x_i)$, by the definition, $frame(x_i) \equiv \Box(more \rightarrow \bigcirc lbf(x_i))$, we have,

$$\Phi(frame(x_i)) \stackrel{\text{def}}{=} \Box(more \rightarrow \bigcirc \Phi(lbf(x_i)))$$

6. For $p \wedge q$, $\Phi(p \wedge q) \stackrel{\text{def}}{=} \Phi(p) \wedge \Phi(q)$;
7. For $p \vee q$, $\Phi(p \vee q) \stackrel{\text{def}}{=} \Phi(p) \vee \Phi(q)$;
8. For $\bigcirc p$, $\Phi(\bigcirc p) \stackrel{\text{def}}{=} \bigcirc \Phi(p)$;
9. For $\Box p$, $\Phi(\Box p) \stackrel{\text{def}}{=} \Box \Phi(p)$;
10. For $p; q$, $\Phi(p; q) \stackrel{\text{def}}{=} \Phi(p); \Phi(q)$;
11. For *if b then p else q*, by the definition, *if b then p else q* $\equiv b \wedge p \vee \neg b \wedge q$, we have,

$$\Phi(\textit{if } b \textit{ then } p \textit{ else } q) \stackrel{\text{def}}{=} p_b \wedge \Phi(p) \vee \neg p_b \wedge \Phi(q)$$

12. For $(p_1, ..., p_m)prj\ q$, $\Phi((p_1, ..., p_m)prj\ q) \stackrel{\text{def}}{=} (\Phi(p_1), ..., \Phi(p_m))prj\ \Phi(q)$;
13. For *while b do p*, by the definition, *while b do p* $\equiv (p \wedge b)^* \wedge \Box(empty \rightarrow \neg b)$, we have,

$$\Phi(\textit{while } b \textit{ do } p) \stackrel{\text{def}}{=} (\Phi(p) \wedge P_b)^* \wedge \Box(empty \rightarrow \neg p_b)$$

14. For $p||q$, by the definition, $p||q \equiv p \wedge (q; true) \vee q \wedge (p; true)$, we have,

$$\Phi(p||q) \stackrel{\text{def}}{=} \Phi(p) \wedge (\Phi(q); true) \vee \Phi(q) \wedge (\Phi(p); true)$$

15. For $await(b)$, by the definition, $await(b) \equiv (frame(x_1) \wedge ... \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$,

$$\Phi(await(b)) \stackrel{\text{def}}{=} (\Phi(frame(x_1)) \wedge ... \wedge \Phi(frame(x_h))) \wedge \Box(empty \leftrightarrow p_b)$$

16. For $\exists x : q$, since q can be rewritten into its normal form, $q \stackrel{\text{def}}{=} \bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj}$, we have,

$$\begin{aligned}
\Phi(\exists x : q) &\stackrel{\text{def}}{=} \Phi(\exists x : (\bigvee_{i=1}^{l} q_{ei} \wedge empty \vee \bigvee_{j=1}^{t} q_{cj} \wedge \bigcirc q_{fj})) \\
&\equiv \Phi(\bigvee_{i=1}^{l} (\exists x : q_{ei}) \wedge empty \vee \bigvee_{j=1}^{t} (\exists x : q_{cj}) \wedge \bigcirc(\exists x : q_{fj})) \\
&\equiv \bigvee_{i=1}^{l} \Phi(\exists x : q_{ei}) \wedge empty \vee \bigvee_{j=1}^{t} \Phi(\exists x : q_{cj}) \wedge \bigcirc \Phi(\exists x : q_{fj}) \\
&\equiv \bigvee_{i=1}^{l} \bigvee_{k=1}^{n} q_{ei}[d_k/x] \wedge empty \vee \bigvee_{j=1}^{t} \bigvee_{k=1}^{n} q_{cj}[d_k/x] \wedge \bigcirc \Phi(\exists x : q_{fj})
\end{aligned}$$

Thus, for any MSVL program, it can be equivalently expressed by a PPTL formula. □

Notice that in the above proof of 16, we can use $\Phi(\exists x_i p)$ recursively so that a PPTL formula can be obtained. A question one may ask is that this transformation process can terminate? The answer is 'yes' since a simple inductive proof on the structure of p can be made to achieve the conclusion. We omit the details here. In [5], we have proved the finiteness of NFGs of PPTL formulas. Hence, the conclusion also holds for MSVL programs since any MSVL program can be equivalently expressed by a PPTL formula.

Formal Analysis of the Bakery Protocol with Consideration of Nonatomic Reads and Writes

Kazuhiro Ogata and Kokichi Futatsugi

School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
{ogata,kokichi}@jaist.ac.jp

Abstract. The bakery protocol is the first real solution of the mutual exclusion problem. It does not assume any lower mutual exclusion protocols. The bakery protocol has been often used as a benchmark to demonstrate that proposed verification methods and/or tools are powerful enough. But, the true bakery protocol has been rarely used. We have formally proved that the protocol satisfies the mutual exclusion property. The proof is mechanized with CafeOBJ, an algebraic specification language, in which state machines as well as data types can be specified. Nonatomic reads and writes to shared variables are formalized by representing an assignment to a shared variable with multiple atomic transitions. Our formal model of the protocol has states in which a shared variable is being modified. A read to the variable in such states obtains an arbitrary value, which is represented as a CafeOBJ term.

Keywords: CafeOBJ, invariant property, mutual exclusion, observational transition system (OTS), verification.

1 Introduction

The mutual exclusion problem is one of the classic but still important problems in computer science. The problem was originally raised and solved by Edsger Dijkstra in 1965 [1]. Many solutions have been proposed since then. But, it took nine years to solve the problem in the true sense of the word. The first real solution is the bakery protocol proposed by Leslie Lamport in 1974 [2]. All mutual exclusion protocols before the bakery protocol assume lower mutual exclusion protocols. On the other hand, the bakery protocol does not.

The bakery protocol has been often used as a benchmark to demonstrate that proposed verification methods and/or tools, among which are [3,4,5], are powerful enough. But, the true bakery protocol has been rarely used. A simplified version of the bakery protocol has been often used such that the simplified version assumes that a read and a write to a shared variable are performed exclusively.

Lamport gives an informal proof [2] that the protocol satisfies some properties and more rigorous proofs [6,7,8] that the protocol satisfies the mutual exclusion property. The proofs do not assume any atomicity. But, they do not seem to have been mechanized.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 187–207, 2008.

We describe a fully formal and mechanized proof that the protocol satisfies the mutual exclusion protocol. We have faithfully made an abstract model of the protocol as much as the formal method used can. Our abstract model respects nonatomic reads and writes to shared variables, namely that those reads and writes can overlap. Nonatomic reads and writes to shared variables are formalized by representing an assignment to a shared variable with multiple atomic transitions. Our abstract model has states in which a shared variable is being modified. A read to the variable in such states obtains an arbitrary value. Our abstract model uses natural numbers, while the bakery protocol uses integers. But, we do not think that this difference is major because sequences of bits can be naturally interpreted as unsigned integers, or natural numbers. We have formally proved that our abstract model satisfies the mutual exclusion property.

The formal method used is the OTS/CafeOBJ method [9,10]. Observational transition systems (OTSs) are used as abstract models, CafeOBJ [11], an algebraic specification language, is used to specify OTSs and properties to verify, and the CafeOBJ system is used as an interactive proof assistant. We also describe some specification and verification techniques used in the case study, which can be used for not only the OTS/CafeOBJ method but also other formal methods based on algebraic specifications.

The rest of the paper is organized as follows. Section 2 describes the bakery protocol. Sections 3 and 4 introduce CafeOBJ and OTSs, respectively. Section 5 describes how to model the bakery protocol as an OTS, which is specified in CafeOBJ. Section 5 describes the verification. Section 7 discusses some issues on specification and verification. Section 8 mentions some related work. Section 9 concludes the paper.

2 The Bakery Protocol

Many existing mutual exclusion protocols assume atomic instructions, which can be considered lower mutual exclusion protocols. Some assume that read (or load) and write (or store) instructions are atomic [1,12], which implies that a read and a write to a shared variable are performed exclusively. Some assume more complex atomic instructions such as test_and_set and fetch_and_store instructions [13,14]. Unlike those protocols, the bakery protocol does not assume any lower mutual exclusion protocols.

The bakery protocol is an N-process mutual exclusion protocol. The N natural numbers $1, \ldots, N$ are used as the identifications of the N processes, respectively. Fig. 1 shows the protocol in the ALGOL style for each process i. The type of each variable used is integer. $choosing[i]$ and $number[i]$ are shared variables among the N processes. While all processes read the variables, only the process i writes them. j is a local variable to the process i.

The function *maximum* takes N integers and returns one of them, which is not less than the others. The N arguments can be read in any order. $(a, b) < (c, d)$ equals $a < c \vee (a = b \wedge b < d)$.

```
begin integer j;
  L1: choosing[i] := 1;
      number[i] := 1 + maximum(number[1], ..., number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number[j], j) < (number[i], i)
                then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

Fig. 1. The bakery protocol in the ALGOL style

Initially, the process i is in the noncritical section, both *choosing*$[i]$ and *number*$[i]$ are set to 0, and j is an arbitrary integer. We suppose that if the process i enters the critical section, it eventually leaves there, and the process i does not write both *choosing*$[i]$ and *number*$[i]$ in both the critical and noncritical sections.

One of the desired properties the bakery protocol should satisfy is the mutual exclusion property, which means that there exists at most one process in the critical section at any given moment.

3 CafeOBJ

CafeOBJ is an algebraic specification language mainly based on *order-sorted algebras* and *hidden algebras*. Data are specified in terms of the former and state machines are specified in terms of the latter. CafeOBJ has two kinds of sorts: *visible* and *hidden sorts*. Visible and hidden sorts denote carrier sets of order-sorted algebras and hidden algebras, respectively. Elements of visible and hidden sorts are data values and states of state machines, respectively.

There are two kinds of operators: *conventional* and *behavioral operators*. The former are used as data constructors and functions on data, and the latter are used for state machines. The former are also used to represent states of state machines. The latter are classified into *observations* and *actions*. Observations obtain data values that characterize states, and actions change states. A conventional operator f, an observation o and an action a are declared as "`op` f `:` VL `->` S", "`bop` o `:` H VL `->` V" and "`bop` a `:` H VL `->` H", respectively, where VL is a list of visible sorts, S is a visible or hidden sort, V is a visible sort and H is a hidden sort. Conventional operators with no arguments are called constants. An operator can be given attributes such as `assoc`, `comm` and `id:` t, which specify that the operator is associative and commutative, and a term t is an identity of the operator.

There are two kinds of equations: *conventional* and *behavioral equations*. Both can have conditions. A conventional equation says that two data values are equal, and a behavioral equation says that two states are equal in that any observation returns a same data value in the two states and any sequence of actions preserves it. A conventional equation and a behavioral equation are declared as "[c]`eq` l_v `=` r_v [`if` c] ." and "[c]`beq` l_h `=` r_h [`if` c] .", respectively, where l_v and r_v are terms whose sorts are visible, l_h and r_h are terms whose sorts are hidden and c is a term whose sort is `Bool` (see below).

Basic units of CafeOBJ specifications are modules. CafeOBJ provides built-in modules. One of the most important built-in modules is `BOOL` in which propositional logic is specified. `BOOL` is automatically imported by almost every module unless otherwise stated. In `BOOL` and its parent modules, declared are the visible sort `Bool` denoting the set of Boolean values, the constants `true` and `false` of `Bool`, and operators denoting some basic logical connectives. Among the operators are `not_`, `_and_`, `_or_`, `_xor_`, `_implies_` and `_iff_` denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), exclusive disjunction (xor), implication ($\Rightarrow$) and logical equivalence ($\Leftrightarrow$), respectively. An underscore `_` indicates the place where an argument is put such as "b_1 `and` b_2". The operators `_and_`, `_or_` and `_xor_` are given `assoc` and `comm`. The operator `if_then_else_fi` corresponding to the **if** construct in programming languages is also declared. CafeOBJ uses the Hsiang term rewriting system [15] as the decision procedure for propositional logic, which is implemented in `BOOL`. CafeOBJ reduces any term denoting a proposition that is always true (false) to `true` (`false`). More generally, a term denoting a proposition reduces to an exclusively disjunctive normal form of the proposition.

4 Observational Transition Systems (OTSs)

We suppose that there exists a universal state space and each data type used in OTSs is provided. The state space is represented by a hidden sort H and data types are represented by visible sorts such as V_{o1}.

An OTS $\mathcal{S}$ is $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that

- $\mathcal{O}$: A finite set of observers. Each *observer* is represented as an observation declared as "`bop` o `:` H V_{o1} $\ldots V_{om}$ `->` V_o". The equivalence relation ($s_1 =_{\mathcal{S}} s_2$) between two states $s_1, s_2 : H$ is defined as $o(s_1, x_1, \ldots, x_m) = o(s_2, x_1, \ldots, x_m)$ for all $o \in \mathcal{O}$ and all $x_i : V_{oi}$ for $i = 1, \ldots, m$. Observers correspond to variables in the conventional definitions of transition systems.
- $\mathcal{I}$: The set of initial states. An arbitrary initial state is represented by a constant *init* declared as "`op` *init* `: ->` H". Let X_i be a CafeOBJ variable of V_{oi}. The constant *init* is defined with a set of equations. For each observer o, the definition has the equation "`eq` $o(init, X_1, \ldots, X_m)$ `=` *initVal* .", where *initVal* is a term denoting a data value returned by the observer in an arbitrary initial state.
- $\mathcal{T}$: A finite set of transitions. Each *transition* is represented as an action declared as "`bop` t `:` H V_{t1} $\ldots V_{tn}$ `->` H". Each transition preserves

the equivalence relation $=_{\mathcal{S}}$. Each transition t has the condition $c\text{-}t$. If $c\text{-}t(s, y_1, \ldots, y_n)$ does not hold, then $t(s, y_1, \ldots, y_n) =_{\mathcal{S}} s$ for all $s : H$ and all y_j for $j = 1, \ldots, n$. Each transition t is defined with a set of equations. Let S be a CafeOBJ variable of H and Y_j be a CafeOBJ variable of V_{tj}. For each observer o, the definition has the equation "`ceq` $o(t(S, Y_1, \ldots, Y_n), X_1 \ldots, X_m)$ `=` *newVal* `if` $c\text{-}t(S, Y_1, \ldots, Y_n)$ `.`", where *newVal* is a term denoting the data value returned by the observer in the successor state $t(S, Y_1, \ldots, Y_n)$ if the effective condition holds. If t does not change the value returned by o, we may omit the condition and use a nonconditional equation. The definition has one more equation "`cbeq` $t(S, Y_1, \ldots, Y_n)$ `=` S `if not` $c\text{-}t(S, Y_1, \ldots, Y_n)$ `.`", which says that the transition does not change states essentially if the effective condition does not hold.

Given an OTS $\mathcal{S}$, a transition $t \in \mathcal{T}$ and two states $s, s' : H$, if $t(s, b_1, \ldots, b_n) =_{\mathcal{S}} s'$ for some values $b_1 : V_{t1}, \ldots, b_n : V_{tn}$, then we write $s \rightsquigarrow^t_{\mathcal{S}} s'$ and call s' a *t-successor state* of s with respect to (wrt) $\mathcal{S}$. t may be omitted from $s \rightsquigarrow^t_{\mathcal{S}} s'$ and s' may be called a successor state of s wrt $\mathcal{S}$.

Given an OTS $\mathcal{S}$, *reachable states* wrt $\mathcal{S}$ are inductively defined:

- Each $s_0 \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.
- For each $s, s' : H$ such that $s \rightsquigarrow_{\mathcal{S}} s'$, if s is reachable wrt $\mathcal{S}$, so is s'.

Let $\mathcal{R}_{\mathcal{S}}$ be the set of all reachable states wrt $\mathcal{S}$.

Operators whose ranks (types) are H `->` `Bool` are called *state predicates*. Any state predicate p `:` H `->` `Bool` is called *invariant* wrt $\mathcal{S}$ if p holds in all reachable states wrt $\mathcal{S}$, i.e. $\forall s : \mathcal{R}_{\mathcal{S}}. p(s)$.

5 Model and Specification of the Bakery Protocol

We describe the OTS $\mathcal{S}_{\text{Bakery}}$ modeling the bakery protocol, which is specified in CafeOBJ.

5.1 Data Used

Five kinds of data are used in $\mathcal{S}_{\text{Bakery}}$: (1) Boolean values, (2) natural numbers, (3) pairs of natural numbers, (4) sets of natural numbers, and (5) labels.

The built-in module `BOOL` is used for Boolean values. As described in Sect. 3, a Boolean term reduces to an exclusively disjunctive normal form. This is useful because this can check if a given Boolean term is always true (or false). But, it may take too much time for Boolean terms to reduce to their normal forms if the terms have many disjunctions. In addition to `_or_`, `BOOL` has the declaration of another operator `_or-else_` for disjunction. The use of `_or-else_`, instead of `_or_`, can prevent Boolean terms from fully reducing to their exclusive-or normal forms and can save much time. The attributes `assoc` and `comm` are not given to `_or-else_` in `BOOL`. For convenience, those attributes are given to the operator.

The sort `Nat` represents the set of all natural numbers. The constant `0` and the operator `s` denote zero and the successor function. We have the operators

`_=_`, `_<_` and `max`. The first two are the equivalence predicate and the less-than predicate. The third one takes two natural number and returns one that is not less than the other. The attribute `comm` is given to `_=_`.

We have the constant `numOfProcs` of `Nat`, which is the number of processes participating in the protocol. `numOfProcs` corresponds to N in Fig. 1. The predicate `isPid` checks if a given natural number p is used as a process identification, namely that p is greater than 0 and less than or equal to N represented by `numOfProcs`.

We also have the operator `next`. Given a natural number x, the term `next(`x`)` denotes an arbitrary natural number. The operator is used to model an assignment to a shared variable.

The sort `NatPair` represents the set of all pairs of natural numbers. The operator `<_,_>` is the constructor of pairs of natural numbers. We have the operators `_=_` and `_<_`. `<` a`,`b `> = <` c`,`d `>` equals a `=` b `and` b `=` d, and `<` a`,`b `> < <` c`,`d `>` equals a `<` c `or (`a `=` c `and` b `<` d`)`. The attribute `comm` is given to `_=_`.

The sort `NatSet` represents the set of all sets of natural numbers. `Nat` is declared as a sub-sort of `NatSet`, which specifies that a natural number is the singleton set that contains the number. The constant `empty` denotes the empty set and the juxtaposition operator `__` is the constructor of nonempty sets. The attributes `assoc`, `comm` and `id: empty` are given to the juxtaposition operator. We have the operators `_\in_`, `del` and `empty?`. The first checks if a given element is in a given set, the second deletes a given element from a given set if any, and the third checks if a given set is empty. We also have the operator `mkSet`, which takes a natural number n and returns `empty` if n is zero and the term denoting $\{1, \ldots, n\}$ if n is greater than zero.

Labels are used to indicate which parts of the protocol processes are going to execute next. The sort `Label` represents the set of all labels. There are 17 labels, which are represented by the 17 constants `l1`, `l2`, ..., `l15`, `cs`, and `ncs`. We have the operator `_=_`, which is the equivalence predicate on labels. The attribute `comm` is given to `_=_`.

5.2 Definition of Equivalence Predicate on Labels

The operator `_=_` is defined with a set of equations. One equation is "`eq (L = L) = true .`", where `L` is a CafeOBJ variable of `Label`. Given two different labels l_1 and l_2 such as `cs` and `ncs`, we want l_1 `=` l_2 to reduce to `false`. Given one label l and a term x whose final value has not been determined, however, we do not want l `=` x to reduce to either `true` or `false` because x may or may not equal l. One way to define the operator to fulfill the requirement is to declare the equation "`eq (`l_1 `=` l_2`) = false .`" for each pair l_1, l_2 of different labels. But, we need to declare many such equations.

Another solution [16], which does not require to declare many equations, is as follows. We use another sort `RealLabel`, which is declared as a sub-sort of `Label`. The 17 constants `l1`, `l2`, ..., `l15`, `cs`, and `ncs` are declared as those of `RealLabel`. Then, all we have to do is to declare one more equation "`eq (RL1`

`= RL2) = (RL1 == RL2) .`", in addition to the equation "`eq (L = L) = true .`", where `RL1` and `RL2` are CafeOBJ variables of `RealLabel`, and the operator `_==_` is a built-in predicate. The built-in predicate returns `true` if given two terms reduce to a same term and `false` otherwise. That is, it returns `false` even if two terms may represent a same data value. This is why we cannot use `_==_` naively as the equivalence predicate on data values for verification. `RealLabel` is only used to declare the 17 constants. In other places in the specification, `Label` is used.

5.3 Assignments to Shared Variables

Since the Bakery protocol does not assume atomic reads and writes to shared variables, we cannot model an assignment ($x := E;$) to a shared variable as one transition. The assignment is modeled as two or more transitions. The two or more transitions model the following things:

- Zero or more transitions model the calculation of the expression E. If it is not necessary to divide the calculation into multiple steps such that E is a data value such as 0, no transitions are used.
- One transition models the situation that the assignment has started but not finished.
- One transition models the situation that the assignment has finished.

Let `beginWtX` and `endWtX` be the last two transitions, and `x` be the observer with which the value of the variable x is obtained. One of the equations defining `beginWtX` looks like

`ceq x(beginWtX(S,I)) =` *anArbVal* `if c-beginWtX(S,I) .`

and one of the equations defining `endWtX` looks like

`ceq x(endWtX(S,I)) =` *theValOfX* `if c-endWtX(S,I) .`

where `S` is a CafeOBJ variable of a hidden sort denoting the state space, `I` is a CafeOBJ variable of a visible sort denoting process identifications, *anArbVal* is a term denoting an arbitrary value of the visible sort, and *theValOfX* is a term denoting the values obtained by calculating E. In $calS_{\text{Bakery}}$, *anArbVal* is an arbitrary natural number, which is represented as a term `next(`v`)`, where v is a natural number.

When another process `J` than `I` tries to read the variable x in the state `beginWtX(S,I)`, which corresponds to a situation where a `J`'s read to x overlaps an `I`'s write to x , the value obtained by J is arbitrary.

5.4 Choice of Arguments in an Arbitrary Order

The arguments of the *maximum* function can be read in any order. One way to respect the arbitrary choice of arguments and calculate $maximum(number[1], \ldots, number[N])$ is as follows:

1. Set temporary variables *tmp* and m to $\{1, \ldots, N\}$ and 0.
2. If *tmp* is empty, then the calculation is done and m contains the result; otherwise go to 3.
3. Choose and delete an arbitrary number k from *tmp*, set m to $\max(m, k)$, and go to 2.

Three transitions are used to model the calculation: `setTmp`, `checkLC` and `findMax`. Let `tmp` and `m` be the observers with which the values of *tmp* and m are obtained. Let `step1`, `step2` and `step3` denote the locations corresponding to the three steps described above, respectively. Let `step4` denotes the location to which the process moves after finishing the calculation. Let `pc` be the observer that returns the location where the process is.

Some of the equations defining `setTmp` look like

```
ceq pc(setTmp(S)) = step2 if c-setTmp(S) .
ceq tmp(setTmp(S)) = mkSet(numOfProcs) if c-setTmp(S) .
ceq m(setTmp(S)) = 0 if c-setTmp(S) .
```

some of the equations defining `checkLC` look like

```
ceq pc(checkLC(S)) = (if empty?(tmp(S)) then step4 else step3 fi)
    if c-checkLC(S) .
eq tmp(checkLC(S)) = tmp(S) .
eq m(checkLC(S)) = m(S) .
```

and some of the equations defining `findMax` look like

```
ceq pc(findMax(S,K)) = step2 if c-findMax(S,K) .
ceq tmp(findMax(S,K)) = del(tmp(S),K) if c-findMax(S,K) .
ceq m(findMax(S,K)) = max(m(S),K) if c-findMax(S,K) .
```

where `S` is a CafeOBJ variable of a hidden sort denoting the state space, `K` is a CafeOBJ variable of `Nat`, `c-setTmp(S)` is `pc(S) = step1`, `c-checkLC(S)` is `pc(S) = step2`, and `c-findMax(S,K)` is `pc(S) = step3 and K \in tmp(S)`. The transition `findMax` arbitrarily chooses a natural number `K` that is in `tmp(S)`.

5.5 Observers and Transitions

Seven observers are used, which are declared as follows:

```
bop pc     : Sys Nat -> Label   bop choosing : Sys Nat -> Nat
bop number : Sys Nat -> Nat     bop j        : Sys Nat -> Nat
bop tmp    : Sys Nat -> NatSet  bop m        : Sys Nat -> Nat
bop rand   : Sys -> Nat
```

`pc` returns the location where a given process is in a given state. `choosing`, `number` and `j` correspond to the variables found in the bakery protocol. `tmp` and `m` are used to model the calculation of $maximum(number[1], \ldots, number[N])$. `rand` returns an arbitrary natural number, which is used to model assignments to shared variables.

l1: `beginWtCh1` $\longrightarrow$ L1: $choosing[i] := 1$;
l2: `endWtCh1`

l3: `setTmp` $\longrightarrow$ $number[i] := 1 + maximum(number[1], \ldots, number[N])$;
l4: `checkLC1`
l5: `findMax`
l6: `beginWtNum1`
l7: `endWtNum1`

l8: `beginWtCh2` $\longrightarrow$ $choosing[i] := 0$;
l9: `endWtCh2`

l10: `setJ` $\longrightarrow$ **for** $j = 1$ **step** 1 **until** N **do**
l11: `checkLC2` **begin**
l12: `checkCh` L2: **if** $choosing[j] \neq 0$ **then goto** L2;
l13: `checkNum` L3: **if** $number[j] \neq 0$ **and** $(number[j], j) < (number[i], i)$
then goto L3;
end;

cs: `execCS` $\longrightarrow$ critical section;

l14: `beginWtNum2` $\longrightarrow$ $number[i] := 0$;
l15: `endWtNum2`

ncs: `execNCS` $\longrightarrow$ nonciritcal section;

ncs: `tryCS` $\longrightarrow$ **goto** L1;

Fig. 2. Correspondence between transitions and the protocol

18 transitions are used, which are declared as follows:

```
bop beginWtCh1  : Sys Nat -> Sys   bop endWtCh1  : Sys Nat -> Sys
bop setTmp      : Sys Nat -> Sys   bop checkLC1  : Sys Nat -> Sys
bop findMax     : Sys Nat Nat -> Sys
bop beginWtNum1 : Sys Nat -> Sys   bop endWtNum1 : Sys Nat -> Sys
bop beginWtCh2  : Sys Nat -> Sys   bop endWtCh2  : Sys Nat -> Sys
bop setJ        : Sys Nat -> Sys   bop checkLC2  : Sys Nat -> Sys
bop checkCh     : Sys Nat -> Sys   bop checkNum  : Sys Nat -> Sys
bop execCS      : Sys Nat -> Sys
bop beginWtNum2 : Sys Nat -> Sys   bop endWtNum2 : Sys Nat -> Sys
bop execNCS     : Sys Nat -> Sys   bop tryCS     : Sys Nat -> Sys
```

Figure2 shows the correspondence between the 18 transitions and the bakery protocol in the ALGOL style. The first 16 labels represented by the 16 constants `l1`, `l2`, ..., `l15` and `cs` correspond to the first 16 transitions, respectively. The label `ncs` correspond to both `execNCS` and `tryCS`.

```
-- setTmp
eq c-setTmp(S,I) = (pc(S,I) = l3) .
ceq pc(setTmp(S,I),J)
    = (if I = J then l4 else pc(S,J) fi) if c-setTmp(S,I) .
eq choosing(setTmp(S,I),J) = choosing(S,J) .
eq number(setTmp(S,I),J)   = number(S,J) .
eq j(setTmp(S,I),J)        = j(S,J) .
ceq tmp(setTmp(S,I),J)
    = (if I = J then mkSet(numOfProcs) else tmp(S,J) fi)
    if c-setTmp(S,I) .
ceq m(setTmp(S,I),J)
    = (if I = J then 0 else m(S,J) fi) if c-setTmp(S,I) .
eq rand(setTmp(S,I))       = rand(S) .
bceq setTmp(S,I)           = S  if not c-setTmp(S,I) .

-- checkLC1
eq c-checkLC1(S,I) = (pc(S,I) = l4) .
ceq pc(checkLC1(S,I),J)
    = (if I = J then (if empty?(tmp(S,I)) then l6 else l5 fi)
       else pc(S,J) fi) if c-checkLC1(S,I) .
eq choosing(checkLC1(S,I),J) = choosing(S,J) .
eq number(checkLC1(S,I),J)   = number(S,J) .
eq j(checkLC1(S,I),J)        = j(S,J) .
eq tmp(checkLC1(S,I),J)      = tmp(S,J) .
eq m(checkLC1(S,I),J)        = m(S,J) .
eq rand(checkLC1(S,I))       = rand(S) .
bceq checkLC1(S,I)           = S if not c-checkLC1(S,I) .

-- findMax
eq c-findMax(S,I,K) = (pc(S,I) = l5 and K \in tmp(S,I)) .
ceq pc(findMax(S,I,K),J)
    = (if I = J then l4 else pc(S,J) fi) if c-findMax(S,I,K) .
eq choosing(findMax(S,I,K),J) = choosing(S,J) .
eq number(findMax(S,I,K),J)   = number(S,J) .
eq j(findMax(S,I,K),J)        = j(S,J) .
ceq tmp(findMax(S,I,K),J)
    = (if I = J then del(tmp(S,I),K) else tmp(S,J) fi)
    if c-findMax(S,I,K) .
ceq m(findMax(S,I,K),J)
    = (if I = J then max(m(S,I),number(S,K)) else m(S,J) fi)
    if c-findMax(S,I,K) .
eq rand(findMax(S,I,K))       = rand(S) .
bceq findMax(S,I,K)           = S if not c-findMax(S,I,K) .
```

Fig. 3. Definitions of transitions (1)

`setTmp`, `checkLC1` and `findMax` correspond to $maximum(number[1], \ldots, number[N])$. `beginWtNum1` and `endWtNum1` correspond to the assignment of the value obtained by incrementing the result of the calculation to $number[i]$.

```
-- beginWtNum1
eq c-beginWtNum1(S,I) = (pc(S,I) = l6) .
ceq pc(beginWtNum1(S,I),J)
    = (if I = J then l7 else pc(S,J) fi) if c-beginWtNum1(S,I) .
eq choosing(beginWtNum1(S,I),J) = choosing(S,J) .
ceq number(beginWtNum1(S,I),J)
    = (if I = J then rand(S) else number(S,J) fi) if c-beginWtNum1(S,I) .
eq j(beginWtNum1(S,I),J)        = j(S,J) .
eq tmp(beginWtNum1(S,I),J)      = tmp(S,J) .
eq m(beginWtNum1(S,I),J)        = m(S,J) .
ceq rand(beginWtNum1(S,I))      = next(rand(S)) if c-beginWtNum1(S,I) .
bceq beginWtNum1(S,I)           = S if not c-beginWtNum1(S,I) .

-- endWtNum1
eq c-endWtNum1(S,I) = (pc(S,I) = l7) .
ceq pc(endWtNum1(S,I),J)
    = (if I = J then l8 else pc(S,J) fi) if c-endWtNum1(S,I) .
eq choosing(endWtNum1(S,I),J) = choosing(S,J) .
ceq number(endWtNum1(S,I),J)
    = (if I = J then s(m(S,I)) else number(S,J) fi) if c-endWtNum1(S,I) .
eq j(endWtNum1(S,I),J)        = j(S,J) .
eq tmp(endWtNum1(S,I),J)      = tmp(S,J) .
eq m(endWtNum1(S,I),J)        = m(S,J) .
eq rand(endWtNum1(S,I))       = rand(S) .
bceq endWtNum1(S,I)           = S if not c-endWtNum1(S,I) .
```

Fig. 4. Definitions of transitions (2)

`setJ` corresponds to the assignment of 1 to the process i's local variable j. `checkLC2` corresponds to the loop termination check. `checkCh` and `checkNum` correspond to the first and second conditional statements in the inner loop of the protocol, respectively.

5.6 Definitions of Transitions

We cannot show all equations defining the 18 transitions due to the space limitation. We show the equations defining `setTmp`, `checkLC1`, `findMax`, `beginWtNum1` and `endWtNum1` in Fig.3 and Fig.4. Lines starting with "`--`" are comments.

6 Verification Based on the Specification

We describe the mechanized proof that $\mathcal{S}_{\text{Bakery}}$ satisfies the mutual exclusion property based on the specification of $\mathcal{S}_{\text{Bakery}}$. The proof is conducted by writing proof socres in CafeOBJ and executing them with the CafeOBJ system. Proof scores are proofs or proof plans written in an algebraic specification language such as CafeOBJ.

6.1 Formalization of the Mutual Exclusion Property

The mutual exclusion property can be stated as there is at most one process in the critical section at any given moment. This can be rephrased as if there are processes in the critical section, then they are identical. The property is formalized as follows:

```
eq inv1(S,P,Q) = (isPid(P) and isPid(Q) and
     pc(S,P) = cs and pc(S,Q) = cs implies P = Q) .
```

Since $\mathcal{S}_{\text{Bakery}}$ does not explicitly disallow processes whose identifications are not in $\{1, \ldots, N\}$ to participate in the protocol, we need to have `isPid(P) and isPid(Q)` as part of the premises.

What to do is to prove `inv1(S,P,Q)` for all reachable states `S` and all natural numbers `P` and `Q`. The proof is done by induction on the structure of the reachable state space. Then, we declare the constant `istep1` of `Bool`, which is defined as "`eq istep1 = inv1(s,p,q) implies inv1(s',p,q) .`", where `s` is a constant of `Sys` denoting an arbitrary state, `s'` is a constant of `Sys` denoting an arbitrary successor state of `s`, and `p` and `q` are constants of `Nat` denoting arbitrary natural numbers. We suppose that the importation of a module, say `ISTEP`, makes those operators, equations and constants available.

6.2 Lemmas of the Verification

The verification needs to prove that 12 more state predicates are invariant wrt $\mathcal{S}_{\text{Bakery}}$. The 12 state predicates are shown in Table 1. The predicates `inWS2`, `inWS&CS`, `inCM` and `inZS` are defined as follows:

Predicate	Definition
`inWS1(L)`	`L = l8 or-else L = l9 or-else L = l10`
`inWS2(L)`	`L = l11 or-else L = l12 or-else L = l13`
`inWS(L)`	`inWS1(L) or-else inWS2(L)`
`inWS&CS(L)`	`inWS(L) or-else L = cs or-else L = l14`
`inCM(L)`	`L = l4 or-else L = l5 or-else L = l6 or-else L = l7`
`inZS(L)`	`L = ncs or-else L = l1 or-else L = l2 or-else L = l3 or-else L = l4 or-else L = l5 or-else L = l6`

Instead of `_or_`, `_or-else_` is used to prevent Boolean terms from fully reducing to their exclusive-or normal forms.

6.3 Proof Score of `inv1`

Let us take a close look at the protocol to check which transitions preserve `inv1` and which do not seem. In the induction case for a transition that does not seem to preserve `inv1`, we may need lemmas.

All transitions except for `checkLC2` preserve `inv1`. This is because they change `pc(`s`,`p`)` for some process identification p to a value that is different from `cs` or do not change the value returned by any observer. In the former, the change makes the premise of `inv1` false, namely making `inv1` true. In the latter, `inv1` is clearly preserved. We still need to prove the induction case for each transition. But, such

Table 1. The lemmas used in the verification

Lemma	Definition
inv2(S,P,J)	(isPid(P) and inWS2(pc(S,P)) and 0 < J and J < j(S,P) and inWS&CS(pc(S,J)) and not(J = P)) implies < number(S,P),P > < < number(S,J),J >
inv3(S,P,J)	(isPid(P) and pc(S,P) = cs and 0 < J and J < s(numOfProcs) and inWS&CS(pc(S,J)) and not(J = P)) implies < number(S,P),P > < < number(S,J),J >
inv4(S,P)	inWS&CS(pc(S,P)) implies 0 < number(S,P)
inv5(S,P,Q)	(isPid(P) and isPid(Q) and inCM(pc(S,Q)) and (inWS2(pc(S,P)) or-else pc(S,P) = cs) and not(P \in tmp(S,Q)) and Q < j(S,P) and not(P = Q)) implies (number(S,P) = m(S,Q) or number(S,P) < m(S,Q))
inv6(S,P)	(pc(S,P) = l6 or-else pc(S,P) = l7) implies empty?(tmp(S,P))
inv7(S,P)	inWS2(pc(S,P)) implies (j(S,P) = s(numOfProcs) or-else j(S,P) < s(numOfProcs))
inv8(S,P)	pc(S,P) = cs implies j(S,P) = s(numOfProcs)
inv9(S,P)	(isPid(P) and isPid(j(S,P)) and inCM(pc(S,j(S,P))) and pc(S,P) = l13 and not(P \in tmp(S,j(S,P))) and not(P = j(S,P)) and (number(S,j(S,P)) = 0 or-else not(< number(S,j(S,P)),j(S,P) > < < number(S,P),P >))) implies (number(S,P) = m(S,j(S,P)) or number(S,P) < m(S,j(S,P)))
inv10(S,P)	(inCM(pc(S,j(S,P))) and pc(S,P) = l12 and choosing(S,j(S,P)) = 0) implies (number(S,P) = m(S,j(S,P)) or number(S,P) < m(S,j(S,P)))
inv11(S,P)	(inCM(pc(S,P)) or-else pc(S,P) = l3) implies not(choosing(S,P) = 0)
inv12(S,P)	inZS(pc(S,P)) implies number(S,P) = 0
inv13(S,P)	(pc(S,P) = l12 or-else pc(S,P) = l13) implies j(S,P) < s(numOfProcs)

a thought experiment makes it clear that only case splitting can discharge the induction case for all transitions except for `checkLC2` but we may also need lemmas for `checkLC2`.

Let us consider the following *proof passage* (a fragment of a proof score):

```
open ISTEP
-- arbitrary values
  op i : -> Nat .
-- assumptions
  -- eq c-checkLC2(s,i) = true .
  eq pc(s,i) = l11 .
  --
  eq p = i .
  eq (q = i) = false .
  eq j(s,i) = s(numOfProcs) .
  eq pc(s,q) = cs .
-- successor state
```

```
  eq s' = checkLC2(s,i) .
-- check
  red istep1 .
close
```

The command `open` makes a temporary module in which a given module (`ISTEP` in this case) is imported, and the command `close` destroys such a module. The constant `i` is used to denote an arbitrary natural number. The constant `s` is used to denote an arbitrary state in which the first five equations hold. The last equation says that `s'` is an arbitrary `checkLC2`-successor state of `s`. The command `red` reduces a given term.

Since `pc(s',p)` equals `cs`, `pc(s',q)` equals `cs` and `p` does not equal `q`, if `p` and `q` are used as process identifications and `s` is reachable, then `inv1` is not invariant wrt $\mathcal{S}_{\text{Bakery}}$. We need to conjecture lemmas to discharge the proof passage.

A close inspection of the bakery protocol allows us to conjecture the following two statements: for all reachable states s, all process identifications p and all natural numbers q,

1. If `pc(`s`,`p`)` is `l1`, `l2` or `l3`, `pc(`s`,`q`)` is `l8`, ..., `cs` or `l14`, q is not p, and q is greater than 0 and less than `j(`s`,`p`)`, then `< number(`s`,`p`),`p `>` is less than `< number(`s`,`q`),`q `>`.
2. If `pc(`s`,`p`)` is `cs`, `pc(`s`,`q`)` is `l8`, ..., `cs` or `l14`, q is not p, and q is greater than 0 and less than `s(numOfProcs)`, then `< number(`s`,`p`),`p `>` is less than `< number(`s`,`q`),`q `>`.

Both statements roughly say that if it has been decided that p has high priority over q when p is in the inner loop of the protocol, the situation lasts while p is in the inner loop or in the critical section. They corresponds to `inv2` and `inv3` in Table1, respectively.

Instead of `istep1`, we reduce `inv2(s,p,q) and inv3(s,q,p) implies istep1` in the proof passage, whose results is `true`. This is because if both `p` and `q` are process identifications, `inv2(s,p,q) and inv3(s,q,p)` is equivalent to `< number[p],p > < < number[q],q > and < number[q],q > < < number[q],q >`, which reduces to `false`.

In the proof score of `inv1`, there is one more proof passage, which uses `inv2` and `inv3`, to discharge the proof passage:

```
open ISTEP
  op i : -> Nat .
  eq pc(s,i) = l11 .
  eq (p = i) = false .
  eq q = i .
  eq j(s,i) = s(numOfProcs) .
  eq pc(s,p) = cs .
  eq s' = checkLC2(s,i) .
  red inv2(s,q,p) and inv3(s,p,q) implies istep1 .
close
```

Comments are omitted. Any other proof passages do not use any lemmas.

6.4 Proof Score of inv2

All transitions except for endWtNum1, setJ and checkNum preserve inv2. This is because although they may change pc(*s*,*p*) for some process identification, they do not change the truth value of inv2. setJ may change pc(*s*,*p*) for some process identification to l11 from l10, but if it does, it sets j(*s*,*p*) to s(0), which makes the premise of inv2 false and then makes inv2 true. We may need lemmas in the induction case for endWtNum1 and checkNum.

Let us consider the following proof passage:

```
open ISTEP
-- arbitrary values
  op i : -> Nat .
-- assumptions
  -- eq c-endWtNum1(s,i) = true .
  eq pc(s,i) = l7 .
  --
  eq (p = i) = false .
  eq j = i .
  eq number(s,p) < s(m(s,i)) = false .
  eq (number(s,p) = s(m(s,i)) and p < i) = false .
  eq 0 < p = true .
  eq p < s(numOfProcs) = true .
  eq (pc(s,p) = l11 or-else pc(s,p) = l12
                    or-else pc(s,p) = l13) = true .
  eq 0 < i = true .
  eq i < j(s,p) = true .
  eq (j(s,p) = s(numOfProcs) or-else j(s,p) = numOfProcs
                             or-else j(s,p) < numOfProcs) = true .
  eq i < s(numOfProcs) = true .
  eq tmp(s,i) = empty .
-- successor state
  eq s' = endWtNum1(s,i) .
-- check
  red istep2 .
close
```

CafeOBJ returns false for this proof passage. If inv2 is invariant wrt $\mathcal{S}_{\text{Bakery}}$, an arbitrary state s characterized by the first 13 equations must be unreachable. We need to find lemmas to show that s is unreachable.

A close inspection of the bakery protocol allows us to conjecture the following statement: for all reachable states s and all process identifications p, q,

1. If pc(*s*,*p*) is l1, l2, l3 or cs, pc(*s*,*q*) is l4, ..., l7, q is not p, q is less than j(*s*,*p*) and p is not in tmp(*s*,*p*), then number(*s*,*p*) is less than or equal to m(*s*,*q*).

The statement roughly says that if it has been decided that p has high priority over q when p is in the inner loop of the protocol and q is calculating

$maximum(number[1], \ldots, number[N])$, the situation lasts while p is in the inner loop or in the critical section and q is calculating the expression. The statement corresponds to `inv5` in Table1.

`inv5(s,p,j)` reduces to `false` in the proof passage, which means that if `inv5` is invariant wrt $\mathcal{S}_{\text{Bakery}}$, an arbitrary state `s` characterized by the first 13 equations in the proof passage is unreachable. Therefore, `inv5` discharges the proof passage.

In addition to `inv5`, the induction case where `endWtNum1` is considered needs `inv6` and `inv7` shown in Table1. The induction case for `checkNum` needs `inv4` shown in Table1. The induction case where other transitions are considered does not need any lemmas.

6.5 Other Proof Scores

All the other lemmas except for `inv10` are proved by induction on the structure of the reachable state space. A simple logical calculation deduces `inv10` from `inv11`, which is also done by writing a proof score. The proofs of some lemmas also need lemmas, which are as follows: (1) `inv3`: `inv2`, `inv6` `inv5` and `inv8`; (2) `inv4`: no lemmas; (3) `inv5`: `inv8`; (4) `inv6`: no lemmas; (5) `inv7`: `inv13`; (6) `inv8`: no lemmas; (7) `inv9`:`inv10` and `inv12`; (8) `inv11`: no lemmas; (9) `inv12`: no lemmas; (10) `inv13`: `inv7`. The verification also needs several lemmas on natural numbers.

7 Discussion

7.1 Choice of Arguments in an Arbitrary Order

We have described a way to formalize choice of arguments in an arbitrary order in Subsect.5.4. Another seemingly possible way to do so, which we first came up with, is to use the operator `choose` declared and defined as follows:

```
op choose : NatSet -> Nat
eq choose(X S) = X .
```

where `X` and `S` are CafeOBJ variables of `Nat` and `NatSet`, respectively.

We thought that the equation successfully formalized an arbitrary choice of a natural number `X` among the set of natural numbers in which the natural number was. But, the equation makes all natural numbers identical. This is because since the juxtaposition operator `__` is associative and commutative, `x y xs` equals `y x xs` where `x` and `y` are arbitrary natural numbers and `xs` is an arbitrary set of natural numbers and then `choose(x y xs)` equals `choose(y x xs)`, which leads to the equivalence of `x` and `y` due to the equation.

7.2 Lemmas on Data

The verification also needs lemmas on natural numbers. There are at least two ways to declare lemmas on data such as natural numbers in the OTS/CafeOBJ

method. They can be declared as (1) standard (conditional) equations and (2) Boolean terms. An example of the first solution is "`eq (X < Y and Y < X) = false .`" and an example of the second solution is "`eq natLem7(X,Y,Z) = (X < Y and Y < Z implies X < Z) .`". Both lemmas are used in the verification.

Both solutions have pros and cons. The first solution's good and bad points are as follows:

Good points. Basically lemmas as standard equations can be used automatically by reduction and users do not care about where lemmas should be used.

But, there are some lemmas, which cannot be used automatically by reduction. An example is "`eq X < Z = true if X < Y and Y < Z .`". This is because the variable `Y` in the condition does not occur on the left-hand side of the equation.

Bad points. Lemmas as standard equations may affect confluence and terminating of specifications. We suppose that we use the lemma "`eq X < Y and Y < s(X) = false .`". If we also declare the lemma "`eq Y < s(X) = (Y = X or-else Y < X) .`", the specification becomes nonconfluent. The first lemma should be modified as "`eq X < Y and (Y = X or-else Y < X) = false .`".

The second solution's good and bad points are as follows:

Good points. Lemmas as Boolean terms do not affect confluence and terminating of specifications.

Bad points. Lemmas as Boolean terms are not used automatically by reduction. Users need to care about where lemmas should be used.

Since lemmas should be used explicitly, however, the second solution allows users to understand the reason why lemmas should be used and makes proofs more traceable.

8 Related Work

The bakery protocol has been often used as a benchmark to demonstrate that proposed verification methods and/or tools are powerful enough. Among such methods and tools are [3,4,5].

Mori, et al. [3] proves with a resolution-based theorem prover implemented on top of the CafeOBJ system that a simplified version of the protocol satisfies the mutual exclusion property. They assume that N processes participate in the protocol as we do. Their way to model the protocol is similar to ours. The resolution-based theorem prover could be used to proves that $\mathcal{S}_{\text{Bakery}}$ satisfies the mutual exclusion protocol.

Meseguer, et al. [4] proves with an abstraction method and the Maude LTL model checker [17,18] that a simplified version of the protocol satisfies the mutual exclusion property. They only consider that two processes participate in the protocol. Their verification technique is based on rewriting like ours. But, their

way to specify state machines is different from ours. Their method needs to fix a concrete number, say 2, of processes participating in the protocol. Although it is possible to represent nonatomic reads and writes in their method, it does not seem clear to come up with a good abstraction when nonatomic reads and writes are taken into account.

de Moura, et al. [5] proves with the SAL [19] implementation of k-induction that a simplified version of the protocol satisfies the mutual exclusion property. The implementation uses an SMT-based bounded model checker. They only consider that two processes participate in the protocol. Their way to specify state machines needs to fix a concrete number, say 2, of processes participating in the protocol like the method used in [4]. It does not seem clear to model nonatomic reads and writes because it does not seem clear to express arbitrary values.

All the simplified versions of the protocol assume that a read and a write to a shared variable are performed exclusively. The true bakery protocol has been rarely used.

Lamport gives an informal proof that the bakery protocol satisfies some properties including the mutual exclusion one [2]. He also gives a more rigorous but nonassertional proof that a variant of the bakery protocol satisfies the mutual exclusion property [6]. The nonassertional proof does not assume any atomicity, but uses as axioms some relations between reads and writes to shared variables.

He gives an assertional proof that the bakery protocol satisfies the mutual exclusion protocol [7]. The proof does not assume any atomicity, either. In the proof, Lamport introduced the predicate transformers *win* (the weakest invariant operator) and *sin* (the strongest invariant operator), which generalize *wlp* (the weakest liberal precondition operator) and *sp* (the strongest postcondition operator). Statements such as assignments that constitute the protocol are represented by (nonatomic) operations, which basically consist of atomic operations. But, the proof does not assume what atomic operations constitute the operation denoting each statement of the protocol. The proof revealed the two hidden assumptions that the assignment $(number[i] := 1 + maximum(number[1], \ldots, number[N]))$ sets $number[i]$ (1) positive and (2) greater than $number[j]$, even if it is executed while the value of $number[k]$ is being changed, for $k \neq i, j$.

We assume some atomicity, namely that every transition is atomic. But, we do not make an assumption that reads and writes to shared variables are atomic. In our abstract model of the protocol, the assignment $(number[i] := 1 + maximum(number[1], \ldots, number[N]))$ is represented by five transitions, while it is represented by a nonatomic operation in the Lamport's abstract model. Our proof does not need the two hidden assumptions. This is because our abstract model is more concrete than the Lamport's abstract model. Neither the Lamport's assertional and nonassertional proofs do not seem to have been mechanized, although they could be.

Lamport also uses multiple atomic transitions (or atomic operations) to represent a nonatomic assignment to a shared variable in [8]. He defines a nonatomic assignment of a value v to a shared variable x by the two atomic assignments

$x :=?$ and $x := v$. When x equals ?, a read to x obtains an arbitrary value, which needs to change the semantics of reads to shared variables. He proves that the bakery protocol satisfies some safety and liveness properties with consideration of nonatomic reads and writes. The proof does not need the two hidden assumptions.

Our abstract model is similar to the Lamport's one described in [8]. Our abstract model is written in an algebraic specification language, however, while his is written as a flowchart. The expressiveness of an algebraic specification language allows us to represent an arbitrary natural number as a term. Therefore, we do not need to change the semantics of reads to shared variables.

Another way to define a nonatomic assignment to a shared variable by multiple atomic transitions (or atomic operations) is given in [20]. A nonatomic assignment of a value v to a shared variable x is represented as a nondeterministic program fragment in which x is incremented (and decremented) arbitrarily but finitely many times and finally x is set to v. This solution does not need to change the semantics of reads to shared variables.

9 Conclusion

We have described a fully formal proof that the bakery protocol satisfies the mutual exclusion protocol. The proof has been mechanized with CafeOBJ. The CafeOBJ system has been used as an interactive proof assistant. Nonatomic reads and writes to shared variables have been formalized by representing an assignment to a shared variable with multiple atomic transitions. Our formal model of the protocol has states in which a shared variable is being modified. A read to the variable in such states obtains an arbitrary value, which is represented as a CafeOBJ term.

One piece of our future work is to conduct the verification based on the specification where integers are used instead of natural numbers. Another one is to prove that the protocol satisfies other properties such as the lockout (or starvation) freedom property, which is a liveness property.

References

1. Dijkstra, E.W.: Solution of a problem in concurrent programming control. CACM 8, 569 (1965)
2. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. CACM 17, 453–455 (1974)
3. Mori, A., Futatsugi, K.: Cafeobj as a tool for behavioral system verification. In: Okada, M., Pierce, B.C., Scedrov, A., Tokuda, H., Yonezawa, A. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 2–16. Springer, Heidelberg (2003)
4. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstraction. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 2–16. Springer, Heidelberg (2003)
5. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)

6. Lamport, L.: A new approach to proving the correctness of multiprocess programs. ACM TOPLAS 1, 84–97 (1979)
7. Lamport, L.: win and sin: Predicate transformers for concurrency. ACM TOPLAS 12, 396–428 (1990)
8. Lamport, L.: Proving the correctness of multiprocess programs. IEEE TSE SE-3, 125–143 (1977)
9. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 170–184. Springer, Heidelberg (2003)
10. 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)
11. Diaconescu, R., Futatsugi, K.: CafeOBJ report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, vol. 6. World Scientific, Singapore (1998)
12. Peterson, G.L.: Myths about the mutual exclusion problem. IPL 12, 115–116 (1981)
13. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE TPDS 1, 6–16 (1990)
14. Mellor-Crummery, J.M., Scott, L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM TOCS 9, 21–65 (1991)
15. Hsiang, J., Dershowitz, N.: Rewrite methods for clausal and nonclausal theorem proving. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 331–346. Springer, Heidelberg (1983)
16. Nakamura, M., Futatsugi, K.: On equality predicates in algebraic specification languages. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 381–395. Springer, Heidelberg (2007)
17. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: 4th WRLA. ENTCS, vol. 71. Elsevier, Amsterdam (2004)
18. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. In: Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.) All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
19. 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)
20. Anderson, J.H., Gouda, M.G.: Atomic semantics of nonatomic programs. IPL 28, 99–103 (1988)

Towards Abstraction for DynAlloy Specifications

Nazareno M. Aguirre[1], Marcelo F. Frias[2], Pablo Ponzio[1], Brian J. Cardiff[2], Juan P. Galeotti[2], and Germán Regis[1]

[1] Department of Computer Science, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Argentina
{naguirre,pponzio,gregis}@dc.exa.unrc.edu.ar
[2] Department of Computer Science, FCEyN, Universidad de Buenos Aires and CONICET, Argentina
{mfrias,bcardiff,jgaleotti}@dc.uba.ar

Abstract. DynAlloy is an extension of the Alloy language to better describe state change via actions and programs, in the style of dynamic logic. In this paper, we report on our experience in trying to provide abstraction based mechanisms for improving DynAlloy specifications with respect to SAT based analysis. The technique we employ is based on predicate abstraction, but due to the context in which we make use of it, is subject to the following more specific improvements: *(i)* since DynAlloy's analysis consists of checking partial correctness assertions against programs, we are only interested in the initial and final states of a computation, and therefore we can safely abstract away some intermediate states in the computation (generally, this kind of abstraction cannot be safely applied in model checking), *(ii)* since DynAlloy's analysis is inherently bounded, we can safely rely on the sole use of a SAT solver for producing the abstractions, and *(iii)* since DynAlloy's basic operational unit is the atomic action, which can be used in different parts within a program, we can reuse the abstraction of an action in different parts of a program, which can accelerate the convergence in checking valid properties.

We present the technique via a case study based on a translation of a JML annotated Java program into DynAlloy, accompanied by some preliminary experimental results showing some of the benefits of the technique.

1 Introduction

The increasing dependability of human activities on software systems is leading us to accept that formal methods, once thought to be worthwhile only for critical systems, are actually applicable and even necessary for a wider class of systems. Indeed, there currently exist many tools and projects attempting to bring together formal methods and widely used (less formal) software development notations and methodologies (e.g., the works reported in [17,18,20], to name a few). Two of the main limitations for using formal methods in practice are that they require mathematically trained developers, and that their application often involves the manual manipulations of mathematical expressions, both

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 207–225, 2008.

during modelling (specification) and analysis (e.g., by theorem proving). Alloy [15] is a formal method that attempts to partly overcome these limitations. First, it is based on a simple notation, with a simple relational semantics, which resembles the modelling constructs of less formal object oriented notations, and therefore is easier to learn and use for developers without a strong mathematical background. Second, it offers a completely automated SAT based analysis mechanism, so that, in principle, no manual manipulations of mathematical expressions are necessary for analysing specifications. This is done at the expense of losing certainty, since the Alloy tool cannot guarantee the validity of a property, but only its validity in bounded (usually by a rather small bound) models [15]. Basically, given a system specification and a statement about it, the Alloy tool exhaustively searches for a counterexample of this statement (under the assumptions of the system description), by reducing the problem to the satisfiability of a propositional formula. Since the Alloy language is first-order, the exhaustive search for counterexamples has to be performed up to certain bound k in the number of elements in the universe of the interpretations. Thus, this analysis procedure can be regarded as a *validation* mechanism, rather than a *verification* procedure, since it cannot be used in general to guarantee the absence of counterexamples for a theory. Nevertheless, this analysis mechanism is very useful in practice, since it allows one to discover counterexamples of intended properties, and if none is found, gain confidence about our specifications. This is similar in spirit to testing, since one checks the truth of a statement for a number of cases; however, as explained in [16], the scope of the technique is much greater than that of testing, since the space of cases examined (usually in the order of billions) is beyond what is covered by testing techniques, and it does not require one to manually provide test cases.

Alloy belongs to the class of the so called *model oriented* formal methods. Specifications in Alloy are described as abstract models of software systems. These models are essentially composed of data domains and relations between these domains, much in the style of schemata for data domains and operations in *Z* [21]. As we and other researchers have advocated in the past, this is suitable for building *static* models of software, but it is less appropriate for the dynamics of systems, i.e., for describing executions and their intended properties [9]. This problem has inspired the definition of an extension of Alloy, called DynAlloy [10], that incorporates actions, understood as a general concept associated with state change, and covering composite as well as atomic actions. Actions can be composed as program terms in dynamic logic, i.e., via sequential composition, non deterministic choice, iteration, etc. Moreover, one can provide partial correctness assertions about actions, which the DynAlloy Analyzer then translates into Alloy for their SAT based analysis.

Abstraction is strongly related to simplicity and understandability of models, as well as to their analysability. Usually, models are driven by the first two concerns, i.e., the modeller chooses the level of abstraction in his models trying to faithfully characterise the aspects of software he is interested in, in the most simple way possible. Typically, these models are suitable according to their

understandability, but not the most appropriate with respect to analysis. Indeed, it is generally accepted that abstraction on models is crucial for the successful automated analysis of specifications [3]. In this paper, we are concerned about improving the abstraction of DynAlloy specifications for analysis. We present a technique, which has been implemented in a prototypical tool, that allows us to employ a version of predicate abstraction [12], an abstraction technique successfully used for model checking, on DynAlloy specifications. Because of the context in which we use predicate abstraction, we are able to take advantage of the following improvements:

- Since DynAlloy's analysis is based on checking partial correctness assertions against programs, we are only interested in the initial and final states of a computation, and therefore we can safely abstract away some intermediate states. We take advantage of this situation via a particular automated use of program atomisation [11]. Notice that this kind of improvement cannot be straightforwardly applied in model checking, since the abstraction of intermediate states can lead to missing violations of safety properties.
- Since DynAlloy's analysis is inherently bounded, we can safely rely on the sole use of a SAT solver for producing the abstractions, as well as refining these in a counterexample guided way.
- Since DynAlloy's basic operational unit is the atomic action, which can be used in different parts within a program, we can reuse the abstraction of an action in different parts of a program, which can contribute to accelerating convergence in checking valid properties.

Our presentation will be driven by a model resulting from a translation of Java code. As it will be made clearer later on, this kind of model will enable us to perform some of the transformations required for constructing and refining an abstraction in a more efficient way. It will also enable us to present some preliminary experimental results.

2 A Brief Introduction to Alloy and DynAlloy

In this section we present a brief introduction to Alloy and DynAlloy by means of an example. A thorough description of Alloy can be found in [16].

Our case-study involves a program over sets (of characters) represented as acyclic linked lists, without repeated elements. For representing these, we would need a model of lists. In order to specify lists, a data type for the data stored in the lists is necessary. We can then start by indicating the existence of a set (of atoms) for data, which in Alloy is specified using a *signature*:

```
sig Data { }
```

This is a basic signature. We do not assume any special properties regarding the structure of data. We can now specify what constitutes a list. A list consists of a head (which is a node or may be a null reference), and nodes in turn consist of a data value and an attribute `next` relating the current node to the next one in a linked list:

```
sig List {
   head : Node+NullValue
}
```

```
sig Node {
     val : Data,
     next : Node+NullValue
}
```

According to the semantics of Alloy, fields `val` and `next` are functional relations from `Node` objects to `Data` objects, and from `Node` objects to `Node` objects (or to a constant `NullValue`), respectively. `NullValue` is a signature representing a constant, namely the null reference, defined in the following way:

```
one sig NullValue { }
```

As the previous definitions show, signatures are used to define data domains and their structure. The attributes of a signature denote *relations*. The dot operator '$\cdot$' corresponds to relational composition, generalised to n-ary relations, and having relational image as a special case. So, for example, given a set `L` (not necessarily a singleton) of `Node` atoms, expression `L.next` denotes the relational image of `L` under the relation denoted by `next`. This leads to a relational view of the dot notation that preserves the intuitive navigational reading of dot, as in object orientation.

Using signatures and fields, it is possible to build more complex expressions denoting relations, with the aid of the Alloy operators. Operator $\sim$ denotes relational transposition, $*$ denotes reflexive-transitive closure, and ^ denotes transitive closure of a binary relation. There are also binary operators. Operator $+$ denotes union, & denotes intersection, and dot (.) denotes, as we mentioned before, composition of relations. In all cases, the typing must be adequate. We build formulae from expressions. Binary predicate in checks for inclusion, while $=$ checks for equality. From these (atomic) formulae we define more complex formulae using standard first-order connectives and quantifiers. Negation is denoted by !. Conjunction, disjunction and implication are denoted by &&, || and =>, respectively. Finally, quantifications have the form some $a : A \mid \alpha(a)$ and all $a : A \mid \alpha(a)$. Formulae can be used as axioms that constrain models, called *facts*. For example, the following fact:

```
fact AcyclicLists {
    all l:List, n:Node | n in l.head.(*next) => n !in n.(^next)
}
```

constrains lists to be acyclic. Formulae can also be used in *assertions*, which are properties to be analysed using the Alloy Analyzer. For instance, the following assertion:

```
assert NextInjective {
    all l: List, n1, n2: Node |
        n1+n2 in l.head.*next && n1 != n2 => n1.next != n2.next
}
```

asserts that, for every pair of nodes in a list, if the nodes are different, then their corresponding 'next' nodes are also different. One can also write parameterised formulae, called *predicates*. For example, the following predicate:

```
pred nonEmpty(l: List) {
    l.head != NullValue
}
```

characterises nonempty lists. In order to check an assertion, or ask for models of a predicate, the specifier has to provide *bounds* for the maximum number of elements to be considered for the domains. For instance, the command

```
check NextInjective for 5 but 3 Data, 3 Node
```

checks whether assertion `NextInjective` is true in all possible interpretations with at most five lists, three data items and three nodes. The command

```
run nonEmpty for 1 List but 3 Data, 3 Node
```

asks for models of predicate `nonEmpty` (i.e., nonempty lists) with at most one list, three data items and three nodes. It is also possible to check a formula for an exact number of elements in a domain.

The Alloy Analyzer receives as input an Alloy model and the selection of an assertion to be checked. Using the bounds on the data domains, a clever translation converts the Alloy model and the (negation of the) assertion into a propositional formula. A model (in the mathematical logic sense) of the resulting propositional formula is then sought for, using off-the-shelf SAT solvers. If a model is found by the SAT solver, it is converted back into a model of the Alloy specification that refutes the validity of the assertion in the specification. A similar procedure is employed for retrieving models satisfying predicates.

DynAlloy [10] is an extension of the Alloy specification language for describing state change in a more convenient way (compared to the Alloy approach, which uses predicates to specify state change). DynAlloy incorporates the notion of *atomic action* as a basic mechanism for modifying the state (atomic actions are similar to atomic statements in imperative programming languages). Atomic actions are defined by means of preconditions and postconditions, given as Alloy formulae. For instance, atomic actions for retrieving the first element in a list and for removing the front element from a list (usually called `Head` and `Tail`, respectively) may be specified as follows:

```
act Head(l:List, d:Data)
    pre = { l.head != NullValue }
    post = { d' = (l.head).val }

act Tail(l:List)
    pre = { l.head != NullValue }
    post = { l'.head = (l.head).next }
```

The primed variables `d'` and `l'` in the specification of actions `Head` and `Tail` denote the values of variables `d` and `l` in those states reached *after* the execution of the actions. There is an important point in the definition of the semantics of atomic programs. While actions may modify the value of all variables, we assume

that those variables whose primed versions do not occur in the post condition retain their corresponding input values. Thus, the atomic action `Head` modifies the value of variable `d`, but `l` keeps its initial value.

From atomic actions we can build complex actions, also called *programs*, as follows. If α is an Alloy formula, then $\alpha?$ is a test action (akin to the "assert" construct in the Java programming language). The nondeterministic choice between two (not necessarily atomic) actions a_1 and a_2 is denoted by $a_1 + a_2$, while their sequential composition is denoted by $a_1 ; a_2$. Finally, $*$ iterates actions. As is customary, a partial correctness assertion of the form $\{\alpha\}\, p\, \{\beta\}$ is satisfied if, for every state e that satisfies α, all the states reachable from e through program p satisfy β. For instance, the following is a valid partial correctness assertion for our case study:

```
                { l.head != NullValue }
                  Head(l, d) ; Tail(l)
    { (l.head).val = d' and (l.head).next = l'.head }
```

One of the important characteristics of Alloy is that its specifications can be automatically analysed using the Alloy Analyzer. As we explained before, the Alloy Analyzer allows one to automatically verify if a given assertion holds in all interpretations associated with an Alloy model, with the domain sizes being bounded by user provided bounds. DynAlloy specifications are also subject to automated analysis. In [10], we show how DynAlloy specifications can be translated into Alloy specifications, so that we can indirectly analyse DynAlloy specifications using the Alloy Analyzer. In order to do this, the specifier only needs to provide an extra bound, one to be associated with the maximum number of iterations to be considered.

The case study model employed in this article originates from Java code. Our translation from Java to DynAlloy adopts the object model of JAlloy [14] in order to handle complex data. JAlloy translates Java programs directly to Alloy models. The JAlloy model of signatures `List` and `Node` requires just basic signatures (without fields)

```
sig List { }        sig Node { }
```

and fields are defined as binary relations

```
head : List -> one (Node+NullValue)
val : Node -> one Data
next : Node -> one (Node+NullValue)
```

The modifier "`one`" forces these relations to be total functions. They can be modified by the DynAlloy actions. An action `SetNext`, modelling the update of the value of attribute `next` for a given node, can now be specified as follows:

```
act SetNext(n1,n2:Node+NullValue,next:Node->one(Node+NullValue))
    pre = { n1 != NullValue }
    post = { next' = next ++ (n1 -> n2) }
```

where `++` is relational overriding.

```
/*@ private invariant
  @  (\forall Node n; \reach(this.head).has(n); !\reach(n.next).has(n));
  @*/

/*@ public normal_behavior
   @ assignable theSet;
   @ ensures this.theSet.equals(\old(this.theSet).difference(s.theSet));
   @  also
   @ private normal_behavior
   @   assignable head;
   @*/
public void removeAll(CharSet s) {
     if (this.head != null) {
         Node current = this.head;
         Node prev = null;
         while (current!=null) {
             if (s.isMember(current.value)) {
                 if (prev!=null) prev.next = current.next;
                 else this.head = current.next;
             } else prev = current;
             current = current.next;
   }  }  }
```

Fig. 1. JML-Annotated code to be analysed in this article

3 From JML–Annotated Java Code to DynAlloy

Finding the right case study for analysing our technique is a difficult task. Seeking for such an appropriate case study, which would allow for a controlled increase of code size, and to check some non trivial properties of the code under consideration, we decided our case study to be a Java program solving the following simple problem:

> Given a linked list l (holding characters as information) and a set S of characters as input, remove from l all nodes holding elements in S.

The actual code, including the corresponding JML annotations, is provided in Fig. 1. This program, although simple, is in our opinion fairly adequate, since the size of code can be increased in a controlled way by unrolling the loop required for traversing the list as many times as deemed appropriate. Moreover, it also allows us to check properties such as that the representation invariant, saying that lists are acyclic, is preserved by this program, as well as checking other related properties, such as that the elements removed are no longer part of the list. Notice that expressing the former (see Fig. 1) requires quantification and reachability predicates, which are constructs hard to analyse for most analysis techniques.

In this section we will provide some details regarding how the translation from annotated Java code to DynAlloy is performed, using parts of our case

study. A more thorough description can be found in [8,13]. Our program involves statements for assignment and attribute modification. These are modelled as atomic DynAlloy actions, in the following way:

```
act assign(l:A, r:A)        act SetF(l:A, r:B, F:A->one B)
   pre { true }                pre { l != NullValue }
   post { l' = r }             post { F'=F++(l->r) }
```

In the definition of action `SetF`, the binary relation `F` gets modified by the action. Complex programs are translated as follows:

```
P1 ; P2                    ->   T(P1);T(P2)
if (C) then P1 else P2     ->   (C'?;(P1))+(!C'?;(P2))
while (C) P                ->   (C'?;P)*;!C'?
```

where `C'` is the Alloy translation of predicate `C`. JML assertions are mapped to Alloy formulae. For instance, the first representation invariant in our case study, which constrains lists to be acyclic structures, is translated into the following Alloy formula:

```
all n : Node n | n in this.head.(*next) => !(n in n.(^next)).   (1)
```

This translation is completely automated.

If formula (1) is denoted by `NoCycle(this, head, next, val)`, and `P` is the DynAlloy program obtained from our case study (see Fig. 2), the problem to solve is expressed as the following partial correctness assertion, that we will call `NoCyclePreserved`:

```
{ NoCycle(this, head, next, val) }
                P
{ NoCycle(this', head', next', val') }
```

In order to make the analysis simpler, we will make use of the following DynAlloy atomic action (whose correctness should be checked at a later stage against some implementation):

```
act isMember(result:boolean, s:set Char, c:Char)
    pre { true }
    post { result' = true <=> c in s }
```

4 SAT-Based Predicate Abstraction for DynAlloy Models

We now present the mechanism employed in order to abstract DynAlloy specifications. As we mentioned, the mechanism is based on predicate abstraction and counterexample guided abstraction refinement. We will assume that the reader has some basic acquaintance with the subject as presented in [12,6]. Briefly, standard predicate abstraction works as follows. Given a transition system $P = \langle S, Init, \tau \rangle$, where S is the set of states, $Init$ a formula characterising

```
act RemoveAll(this: List, curr, prev: Node+NullValue, S: set Char,
              value: Node -> one Char, next: Node -> one (Node+NullValue),
              head: List -> one (Node+NullValue))
01. (this.head != NullValue)?;
02. ( assign(prev, NullValue);
03.   assign(curr, this.head);
04.   ( (curr != NullValue)?;
05.     ( ( (
06.           (curr.value in S)?;
07.           (  (prev != NullValue)?;
08.              setNext(prev, curr.next, next)
09.              +
10.              (prev = NullValue)?;
11.              setHead(thisValue, curr.next, head)
12.         )  )
13.         +
14.         ( (curr.value !in S)?;
15.           assign(prev, curr)
16.       ) )
17.       assign(curr, curr.next)
18.     )
19.   )*;
20.   (curr = NullValue)?
21. )
22. +
23. ( (this.head = NullValue)?;
24.   skip
25. )
```

Fig. 2. DynAlloy specification corresponding to program `removeAll`

the set of initial states, and τ a set of transitions (i.e., binary relations over S), one starts by providing some predicates $\phi_1, \phi_2, \ldots, \phi_n$ over S. The main idea is to consider an abstraction Q^A of the lattice $\wp(S)$ of state properties over S, together with two functions $\alpha : \wp(S) \to Q^A$ and $\gamma : Q^A \to \wp(S)$, relating Q^A and $\wp(S)$ in such a way that $\alpha(\gamma(Q^A)) = Q^A$ and, for every $s \in \wp(S)$, $s \subseteq \gamma(\alpha(s))$. That is, the pair $\langle \alpha, \gamma \rangle$ forms a *Galois connection* between Q^A and $\wp(S)$. In predicate abstraction, Q^A has a particular form, it is composed by the monomials over n boolean variables $B_1, B_2, \ldots, B_n$ representing the truth values of $\phi_1, \phi_2, \ldots, \phi_n$, respectively; a monomial is either *true* or *false*, or a conjunction of literals (B_i or $\neg B_i$) in which each B_i appears at most once (positively or negatively). This set clearly forms a lattice, where the atoms (which represent the abstract states) are the canonical monomials, i.e., the monomials in which each B_i appears exactly once. The concretisation function $\gamma : Q^A \to \wp(S)$ is simply defined as $\gamma(s^A) = \{s \in S | s \models s^A[\phi_i/B_i]\}$ whereas the abstraction function is given by:

$$\alpha(s) = \bigwedge_{i \in 1..n} \{B_i | s \models \phi_i\} \wedge \bigwedge_{i \in 1..n} \{\neg B_i | s \models \neg \phi_i\}$$

As explained in [12], this results in a more efficient way of calculating the abstract model from a concrete one.

We would like to provide the above described abstraction mechanism for DynAlloy specifications. As we mentioned, atomic actions are the basic mechanism for characterising state change in DynAlloy, and typically have the following form:

```
act a(s: State)
  pre { pre(s) }
  post { post(s,s') }
```

for some designated state signature `State`. In order to apply predicate abstraction, we need a number of predicates $\phi_1(s), \phi_2(s), \ldots, \phi_n(s)$ over the state signature `State`. In our case, we consider as an initial set of abstraction predicates the individual conjuncts in the postcondition of the assertion, and the conditions extracted from the source code of the program (which appear in test actions in the DynAlloy translation):

```
this.head != NullValue   curr != NullValue
curr.value in S          prev != NullValue
```

Now let us describe how the abstract DynAlloy program is represented. Since in this case we have five predicates (the above four plus the postcondition of the assertion), we can characterise the abstract state space by the following `AState` Alloy signature:

```
sig AState { p0, p1, p2, p3, p4 : Boolean }
```

The idea behind our characterisation of the abstract DynAlloy program is, as the reader might expect, that a particular atom of signature `AState` will represent exactly one abstract state. It is easy to see what are the concrete states associated with an abstract state `s`: those that satisfy exactly those ϕ_i's for which `s.pi` is `true`.

We now need to compute abstract actions characterising the abstract behaviour of each of the concrete atomic actions. Let us first consider atomic actions in isolation. Abstracting the (concrete) precondition *pre* of a given action a is not difficult. We need to decide which are the corresponding elements of the abstract lattice Q^A, i.e., the monomials, better characterising *pre*. This can be done simply by checking which of the ϕ_i's and $\neg\phi_i$'s are implied by *pre*. For postconditions, on the other hand, the process is slightly more complicated. The reason is that, as it is shown in the actions for our DynAlloy program, the postconditions are not state formulae, but *relations* describing how the states previous to the execution of the actions are related to the corresponding states after the execution of the actions. Thus, what we actually need to check, for an atomic action with precondition `pre(s)` and postcondition `post(s,s')`, is the abstract state corresponding to the *strongest postcondition* of `pre(s)` according to `post(s,s')`. We use the Alloy Analyzer in order to check these assertions.

The process just described for computing the abstraction of atomic actions, although correct, generally leaves us with too coarse abstractions, which would produce an important number of spurious counterexamples when checking the abstract program, even when the property being checked is invalid. The reason for this is that this kind of abstraction only takes into account the precondition of the action, and not the information regarding the context in which the action is used. We will use these abstractions as a starting point, and will compute the abstractions using the following more sophisticated approach.

Suppose that we have to check a DynAlloy assertion of the form:

$$\{\ \texttt{pre\_a}\ \}\quad \texttt{P}\quad \{\ \texttt{post\_a}\ \}$$

Notice that every DynAlloy assertion check needs two different bounds, one limiting the size of the domains, and another one bounding loops. Let us consider these to be k_d and k_l, respectively. We will start by unrolling the loops in `P` according to bound k_l, thus obtaining a sequential program P_S, without loops. Our abstraction process will consist of computing an abstract version of P_S's control graph. We will consider, initially, basic abstractions for all atomic actions computed as described above, in terms of their corresponding pre and post conditions and using k_d as a bound. Also, we will compute the abstraction of the precondition `pre_a` of the assertion, also using k_d as a bound. We will then start visiting P_S's computation tree in a depth-first fashion; so, in each step we will choose either a test action (which can be abstracted straightforwardly, since its associated condition is among the abstraction predicates) or an atomic action a. For this, we check whether, for its current abstract precondition (which is not necessarily a canonical monomial), we have already computed its corresponding abstract postcondition. If not, we concretise the current precondition, and compute the abstraction of the corresponding concrete strongest postcondition (for these checks, we also use k_d as a bound). Since P_S's computation tree is finite and acyclic (due to the absence of loops, which we have previously unrolled), this process is guaranteed to terminate. If we reach a final abstract state (a leaf in the computation tree for the abstract version of P_S) in which the abstraction of the postcondition is not satisfied, then we found an abstract counterexample. Notice that the postcondition of the assertion is precisely characterised in the abstract program, since it is included in the set of abstraction predicates. If no abstract counterexample trace is found, then the property has been checked valid within the established bounds.

If an abstract counterexample is found, then we have to check whether it is a spurious one or not. If it is not spurious, the property is invalid, and we find a counterexample as the concretisation of the obtained abstract trace. If, on the other hand, it is spurious, we employ a traditional counterexample guided abstraction refinement as presented in [7]. We find the abstract state in which the spurious counterexample "breaks" (i.e., where it cannot be further concretised), and employ the predicate discovery approach as described in [7].

Let us summarise this process of abstraction. It is composed of the following steps:

1. Take program P and unroll the loops in it according to bound k_l, obtaining as a result a sequential program P_S (both P and P_S are concrete).
2. Compute the abstractions for `pre_a` and `post_a`, and for each of the atomic actions, using the set of abstraction predicates available (initially, these are the conjuncts of the postcondition in the concrete assertion and the conditions in the program).
3. Start the verification process by visiting the computation tree for P_S and computing the corresponding abstract states along the traversal. Here, the abstractions of the actions are used, and more detailed abstract pre- and post-conditions are computed for their definitions when not previously considered abstract preconditions are found).
4. When a final abstract state (a leaf in the computation tree) is reached, there are two possibilities.
   (a) The abstract state satisfies the abstraction of the postcondition. In this case, we continue the visit of the tree. If the whole computation tree has been visited, then the property has been successfully checked (for the given bounds).
   (b) The abstract state violates the abstract postcondition. In this case, we have found an abstract counterexample. We concretise the corresponding trace, and if it corresponds to a concrete counterexample, the property being checked is invalid. If not, the counterexample is spurious, and is used to calculate a new abstraction predicate. We incorporate the new predicate to our set of abstraction predicates, and go back to step 2.

It is important to notice how the above described abstraction mechanism allows us to improve analysability. In the straightforward approach (without abstraction), two variables make the size of the propositional formula resulting from the DynAlloy assertion to be checked grow, namely k_d and k_l. Usually, these two variables in combination make the formula too big to be handled with the available resources. Using the above described abstraction mechanism, the analysis is split into various checks in order to build the abstraction, each of which is only affected by the k_d bound. The only checkings affected by both bounds are the ones corresponding to the concretisation of abstract counterexamples. However, these are generally much simpler than checking the original program, since the trace corresponding to an abstract counterexample is only a sequential program with no branching nor loops (branching has an important negative impact in checking DynAlloy's assertions).

We present below some experimental results regarding the application of the abstraction mechanism just described, in comparison with the straightforward (no abstraction) SAT analysis. However, the described (traditional) abstraction mechanism is not sufficient, and we will need to perform some optimisations in order to gain an acceptable performance for the abstraction based analyses. These optimisations are described in the next section.

5 Improving the Abstraction Based Analysis

In this section we present a few optimisations that we applied to the above described traditional abstraction approach. The contribution, in terms of performance in the analysis, that these optimisations provided are reported later on, in the section on experimental results.

5.1 Program Atomisation

Program atomisation is an abstraction technique for DynAlloy programs that allows us to contribute to scaling analysability up by replacing (arbitrary) complex programs by atomic actions with the same behaviour. The analysis improves because the SAT-solver does not need to look for intermediate valid states matching the program behaviour. Generally, the use of program atomisation is not fully automated, although the way in which we will use it here, a restricted form, enables us to fully automate it. In the context of this article, atomisation will have a great impact, because the removal of intermediate states favours abstraction: intermediate states (when these are temporary) typically cause breaks in spurious counterexamples, and possibly the introduction of further abstraction predicates characterising the corresponding intermediate (temporary) state situations. We will automatically perform atomisation in order to abstract away certain intermediate states, via the following atomisation policy:

> Consider, as a program to be atomised, any maximal sub-path of the control flow graph that does not involve tests.

Notice that program atomisation is applied *before* the program is unrolled according to the bound on iteration.

We still have to provide a method for automatically building the atomisations. Let us consider atomic actions `A1` and `A2` specified as follows:

```
  { Q(s) }        { S(s) }
    A1(s)           A2(s)
{ R(s,s') }     { T(s,s') }
```

The atomisation of the sequential composition `A1 ; A2` is defined as the atomic action `A` specified by:

```
{ Q(s) && (all st | R(s,st) => S(st)) }
                    A
{ some i | R(s,i) && T(i,s') }
```

Notice that `A`'s precondition characterises exactly those states that satisfy the precondition of `A1`, and upon execution of `A1` lead only to states satisfying `A2`'s precondition. On the other hand, the postcondition clearly models the sequential composition of the behaviours of `A1` and `A2`. We are only defining the atomisation for the sequential composition of two actions. If more actions are to be atomised, the process can be iterated. It is straightforward to prove that given actions

```
act atomAssignPrevCurr(prev,prevVal,curr,currVal:Node+NullValue)
   pre = { true }
   post = { prev' = prevVal && curr' = currVal }

act atomSetNextAssCurr(l,r: Node+NullValue,next: Node->one(Node+NullValue),
                       curr,currVal: Node+NullValue)
   pre = { l != NullValue }
   post = { next' = next++(l->r) && curr' = currVal }
```

Fig. 3. Atomisations for program removeAll

A1, A2, A3, atomising first A1 and A2 (and the result with A3) yields an action equivalent to the one resulting from first atomising A2 and A3. Moreover, in order to simplify the resulting atomisation, notice that:

1. Whenever A2's precondition is true, A's precondition reduces to Q(s).
2. If A1 and A2 modify disjoint sets of state variables, we can proceed as follows. Let the state variables $s = S_0 \cup S_1 \cup S_2$, with S_0, S_1, S_2 disjoint, and such that A1 modifies S_1 and A2 modifies S_2. The postcondition then simplifies to

$$\mathtt{R}(S_0 \cup S_1 \cup S_2, S_0 \cup S_1' \cup S_2) \;\&\&\; \mathtt{T}(S_0 \cup S_1' \cup S_2, S_0 \cup S_1' \cup S_2').$$

We perform atomisation at the very beginning, and include simplifications as the ones mentioned to be applied on the resulting atomisations. For our case-study, the original DynAlloy program (see Fig. 2) is atomised using the actions in Fig. 3.

5.2 Detection of Induction

Consider cases in which the program we are trying to check is of the form:

$$P = Init; (a_0 + \cdots + a_n)*$$

with each a_i not necessarily atomic. Moreover, suppose that we check the property under consideration *incrementally*, i.e., we check the property for k_l only after we checked it for all loop bounds smaller than k_l. In these cases, we can consider the following rule in order to "prune" the construction of the computation tree corresponding to P_S in the verification and abstraction process:

> if s is the current abstract state (at some point after initialisation), and the abstraction of a_i leaves us again in the abstract state s, then we can stop building the abstract graph for P_S after the last s, because the abstract states resulting from the last s will necessarily be visited as branches of the first s.

The idea behind this rule is graphically depicted in Figure 4. The fact that this rule is applied only in incremental verifications is crucial, since it corresponds

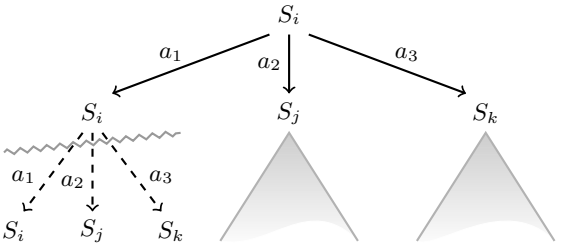


Fig. 4. Graphical description of the pruning rule for detecting convergence in invariant checks.

essentially to performing a kind of iterative deepening visit of the abstract graph, looking for violations of the property (then we only advance in the process if we have not found shorter violations). Also, the fact that atomic actions are repeated in the graph, so that we can use the abstractions produced for these in other parts of the graph, is relevant for this rule. Notice also that, given that we apply program atomisation prior to performing the loop unrolls, it is guaranteed that we are not abstracting away final states, only states which are intermediate within loops.

The case in which we are checking whether the program preserves the representation invariant of sets over linked lists, i.e., whether it leaves the resulting list acyclic and with no repeated elements, corresponds to the above described schema. As we describe below, in the section regarding experimental results, this rule allowed us to find a convergence of the representation invariant property after four loop unrolls. The reason for this is that all abstract final states after four loop unrolls are "already visited" states in the computation tree, enabling us to infer that further loop unrolls cannot lead to violations of the property.

6 Experimental Results

For running the experiments we used a personal computer with a 2Ghz Core 2 Duo processor, with 2GB of RAM, running the Alloy Analyzer 4.1.5 under Ubuntu GNU/Linux 2.6.22, 32 bits.

We attempted to verify the following assertions regarding our case study:

```
NoCyclePreserved:
    all n: Node |
              n in thisV.headV.(*nextV) => n !in n.nextV.(*nextV)

ElementsRemoved:
    all n: Node |
              n in (thisV.headV.(*nextV) - (currV.(*nextV))) =>
                                          n.valueV !in (c1+c2+c3)
```

The first of these, `NoCyclePreserved`, affirms that the list is acyclic, while the second one, `ElementsRemoved`, in combination with the fact that `curr` is null at the end of the algorithm, ensures that the characters in the set to be removed, passed as a parameter, are indeed removed from the list.

Without the use of abstraction, if we unroll the only loop in the program 23 times the analysis of assertion `NoCyclePreserved` exhausts the available memory, causing a run time crash of the Alloy Analyzer (the corresponding CNF formula had 620841 variables, 30768 of which are primary, and a total of 1339236 clauses). In this case, the scopes for signatures `Char` and `Node` were set to 24. Also, if we unroll the loop 51 times, a runtime exception is thrown by the Alloy Analyzer due to insufficient memory to construct the CNF formula to be analysed.

Using the traditional predicate abstraction mechanism described, without the further optimisations, 44 seconds of SAT analysis time were required for our tool to check the property `NoCyclePreserved`. Two new predicates were introduced during the process. Thanks to the fact that our method can take advantage of already calculated abstract actions, performing SAT Solving was only necessary for the first four loop unrolls. For the same assertion, but after applying program atomisation to the original specification, the SAT time was reduced by almost 50%: 25 seconds. The number of predicates and loop unrolls required to discover new abstract actions, though, remained the same. The above described induction detection mechanism allows us to reduce the number of nodes in the abstract execution tree to be visited. Notice that, given a certain number k of loop unrolls, the size of the abstract execution tree for our program is exponential with respect to k. More precisely, in the worst case (when the whole tree has to be visited) we need to visit $1 + \sum_{i=1}^{n} 2^i + \sum_{i=1}^{n-1} 2^i$ nodes for the atomised program and $1 + \sum_{i=1}^{n} 2^i + 2 * \sum_{i=1}^{n-1} 2^i$ nodes for the non atomised case. For instance, if we unroll the loop in the program four times for the atomised case, the abstract tree will be composed of 45 nodes. On the other hand, by employing induction detection the procedure can be terminated after visiting 19 nodes, for the assertion `NoCyclePreserved`. On the other hand, the abstract tree for the nonatomised program, with the loop unrolled four times, will have 59 nodes, and using induction detection our algorithm visited only 30 of them. Furthermore, induction detection allows us to conclude that we can stop the search for abstract counterexamples (of any size), since, as it was discussed in section 5.2, no new states violating the property can appear. As it is shown by this example, this technique can reduce the search state space considerably.

On the other hand, if we want to verify the property `ElementsRemoved` using again the traditional abstraction mechanism, the abstraction process diverges due to excessive introduction of abstraction predicates. Nevertheless, using program atomisation on the model, and introducing the right auxiliary invariants (`prev.next` equals `curr`, `prev` and `curr` are reachable from the head of the list, and the list is acyclic) as abstraction predicates, the process converges again very quickly (9 seconds of SAT time). New atomic actions appeared when verifying the property up to the third iteration, and 13 nodes of the (atomised) abstract computation tree were visited by our algorithm. A total of 8 abstraction

predicates were necessary. It is worth mentioning that `NoCyclesPreserved` was used as an abstract invariant (i.e., a predicate that is valid before and after execution of each abstract action). We did so safely because `NoCyclesPreserved` was previously verified to be a concrete invariant. Notice also that, since the program that we are considering is the same as for the previous case, the number of nodes of the execution tree can be calculated using the same formula (there are 18 nodes in the graph for 3 loop unrolls).

Of course, obtaining the above mentioned auxiliary predicates automatically would require the use of invariant generation techniques. We used the "global invariants" generation mechanism of the STeP tool [2], but in order to do this we had to produce an ad hoc representation of the program in STeP's language SPL. Clearly, this requires further investigation, and is part of our future work.

An important fact to mention regarding the experiments is that, since the DynAlloy specification originates from code, atomic actions essentially correspond to assignments, and therefore these are "invertible", making the calculation of weakest preconditions in sequential programs (when examining abstract counterexamples) very efficient. It is expected that this efficiency will not be preserved when considering other kinds of DynAlloy specifications.

7 Related Work

There exist many tools and approaches applying ideas of predicate abstraction for formal verification, such as for instance the work reported in [6,7]. While our approach is strongly based on Das and Dill's technique for predicate abstraction, it differs from existing tools in that ours is tailored to SAT-solving through Alloy and DynAlloy, as opposed to most other tools, whose associated verification technique is model checking. In [4] SAT-solving is used to construct the abstraction, but the more conventional techniques based on symbolic model checking are used in the remaining parts of the process. With respect to abstraction in the context of Alloy and DynAlloy specifications, the most closely related approaches we know of are the work of Taghdiri [19] on the use of predicate abstraction in JAlloy analysis, and the work of some of the authors of this paper on program atomisation [11]. The work of Taghdiri is different from ours since her abstraction mechanism faithfully represents the code of a Java program, except for the method calls, where abstraction is applied. The abstractions obtained for procedure calls are somehow "reusable" (they can be used for other places in the program where the procedure is also called), as in our approach; however, her mechanism for discovering new predicates is completely different, since in her case the spurious counterexamples are not really abstract runs, but concrete ones where the abstraction is present only in the form of underspecified effects for procedure calls. SATURN [22] is, as our tool, completely based on SAT-solving. The techniques it uses to improve analysability are: program slicing (at the intra-procedural level) and a kind of abstraction called *function summaries* to modularise the analysis at the inter-procedural level. SATURN models programs faithfully (no abstraction is performed).

8 Conclusions and Future Work

We have investigated the application of predicate abstraction for improving the analysis of DynAlloy specifications. We have concentrated in a particular kind of DynAlloy specifications, namely those resulting from a translation from annotated Java code. We plan to exploit the experiences gained in providing abstraction for this kind of DynAlloy specifications in order to provide a mechanism applicable to a wider class of these. We exploited the principal analysis technique associated with Alloy and DynAlloy specifications, SAT based analysis, in order to build, analyse and improve the abstractions in an automated way, based on Das & Dill's algorithm for abstraction refinement [6], complemented primarily with an automated program atomisation mechanism. We have also provided a mechanism for detecting convergence in the unrolling process for a certain schema of DynAlloy programs.

The results of the experiments conducted, based on a case study over a model originated from a JML annotated Java program, show that the improvements performed to the traditional predicate abstraction mechanism had an important impact for one of the properties we checked. The reason is that these constituted an economy in abstraction predicates to be introduced during verification, as well as in the size of the formulae analysed when examining abstract counterexamples for abstraction refinements. However, this is just an initial attempt, and it is clear that we need to develop more case studies.

There are several directions for future work. We are planning to experiment with the use of automated theorem provers to attempt to simplify the formulae introduced in the process of improving abstractions, for eliminating unnecessary quantifiers, amongst other things. Invariant generation techniques, integrated in the approach, will have in our opinion an important positive impact, so this is one of the directions we want to explore. We are also planning to incorporate a differentiated treatment for mutant and non mutant objects in the DynAlloy specifications, in order to make the construction of the abstractions more efficient.

It is also important to mention that our approach corresponds only to intra-procedural analysis. We plan to study the combination of the presented approach with Taghdiri's abstraction for Alloy [19].

References

1. Ball, T., Cook, B., Das, S., Rajamani, S.: Refining Approximations in Software Predicate Abstraction. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988. Springer, Heidelberg (2004)
2. Bjorner, N., Browne, A., Colon, M., Finkbeiner, B., Manna, Z., Sipma, H., Uribe, T.: Verifying Temporal Properties of Reactive Systems: a STeP Tutorial. Formal Methods in System Design 16 (2000)
3. Clarke, E., Grumberg, O., Long, D.: Model checking and abstraction. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(5) (1994)

4. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate Abstraction of ANSI-C Programs using SAT, Technical Report CMU-CS-03-186. Carnegie Mellon University (2003)
5. Cousot, P.: Abstract interpretation. ACM Computing Surveys 28(2) (1996)
6. Das, S., Dill, D.: Successive Approximation of Abstract Transition Relations. In: Proceedings of the IEEE Symposium on Logic in Computer Science LICS 2001. IEEE Computer Society Press, Los Alamitos (2001)
7. Das, S., Dill, D.: Counterexample Based Predicate Discovery in Predicate Abstraction. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517. Springer, Heidelberg (2002)
8. Dennis, G., Chang, F., Jackson, D.: Modular Verification of Code with SAT. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, pp. 109–120 (2006)
9. Frias, M., López Pombo, C., Baum, G., Aguirre, N., Maibaum, T.: Reasoning about static and dynamic properties in alloy: A purely relational approach. ACM Transactions on Software Engineering and Methodology (TOSEM) 14(4) (2005)
10. Frias, M., Galeotti, J.P., López Pombo, C., Aguirre, N.: DynAlloy: upgrading alloy with actions. In: Proceedings of the 27th International Conference on Software Engineering ICSE 2005. ACM Press, New York (2005)
11. Frias, M., Galeotti, J.P., López Pombo, C., Aguirre, N.: Efficient Analysis of DynAlloy Specifications. In: ACM Transactions on Software Engineering and Methodology (TOSEM). ACM Press, New York
12. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254. Springer, Heidelberg (1997)
13. Galeotti, J.P., Frias, M.F.: DynAlloy as a Formal Method for the Analysis of Java Programs. In: Proceedings of IFIP Working Conference on Software Engineering Techniques (SET 2006), Warsaw. Springer, Heidelberg (2006)
14. Jackson, D., Vaziri, M.: Finding Bugs with a Constraint Solver. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), Portland, OR, USA, August 21-24, pp. 14–25. ACM Press, New York (2000)
15. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (ACM TOSEM) 11(2) (2002)
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
17. Kim, S.-K., Carrington, D.: Formalizing the UML Class Diagram Using Object-Z. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723. Springer, Heidelberg (1999)
18. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Transactions on Software Engineering and Methodology (TOSEM) 15(1) (2006)
19. Taghdiri, M.: Inferring Specifications to Detect Errors in Code. In: Proceedings of the 19th International Conference on Automated Software Engineering ASE 2004, Austria (September 2004)
20. Uchitel, S., Chatley, R., Kramer, J., Magee, J.: LTSA-MSC: Tool Support for Behaviour Model Elaboration Using Implied Scenarios. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619. Springer, Heidelberg (2003)
21. Woodcock, J., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice-Hall, Englewood Cliffs (1996)
22. Xie, Y., Aiken, A.: Saturn: A Scalable Framework for Error Detection Using Boolean Satisfiability. ACM-Transactions on Programming Languages and Systems (TOPLAS) (to appear)

Partial Translation Verification for Untrusted Code-Generators*

Matthew Staats and Mats P.E. Heimdahl

Dept. of Comp. Sci. and Eng.
University of Minnesota
{staats,heimdahl}@cs.umn.edu

Abstract. Within the context of model-based development, the correctness of code generators for modeling notations such as Simulink and Stateflow is of obvious importance. If correctness of code generation can be shown, the extensive and often costly verification and validation activities conducted in the modeling domain could be effectively leveraged in the code domain. Unfortunately, most code generators in use today give no guarantees of correctness.

In this paper, we investigate a method of leveraging existing model checking tools to verify the partial correctness of code generated by code generators that offer no guarantees of correctness. We explore the feasibility of this approach through a prototype tool that allows us to verify that Linear Temporal Logic (LTL) safety properties are preserved by C code generators for Simulink models. We find that the approach scales well, allowing us to verify that 55 LTL properties are maintained when generating 12,000+ lines of C code from a large Simulink model.

1 Introduction

Tools translating a source language to a target langauge are probably the most used tools in software development. Of particular interest in this article are code generators for modeling notations extensively used in the critical systems domain, such as Simulink and Stateflow [16,17], and the SCADE tools-suite [8]. In this domain it would be highly desirable if the extensive and often costly verification and validation activities conducted in the modeling domain could be effectively leveraged in the code domain; in other words, if we have have high confidence in the correctness of the model, it would be nice if we could have high confidence in the generated code. Unfortunately, these code generators come with no guarantees of correctness—they are either black-boxes of unknown (though generally good) quality, or have been developed in in-house to address specific needs in terms of generated-code footprint, performance, or applicability to a specific hardware platform.

* This work has been partially supported by NASA Ames Research Center Cooperative Agreement NNA06CB21A, NASA IV&V Facility Contract NNG-05CB16C, and the L-3 Titan Group.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 226–237, 2008.

While interest in the correctness of translators such as traditional compilers and code generators has existed nearly as long as translators themselves [14,18], we still lack the ability to cost-effectively prove the correctness of industrial translators. The automation of correctness proofs is still out of reach, while manual proofs of correctness are both large and difficult (if not entirely infeasible). A different—and potentially more practical—approach to verification is to check individual translations for correctness (as opposed to the translator itself) [22,24,25]. These tools generally rely on a proof of correctness being emitted during the translation process (to be checked by a proof checker). Nevertheless, while this approach is promising, is has not seen adoption in industry. Therefore, for the foreseeable future, we will have to rely on code-generators where we cannot fully trust the generated code.

In this paper, we investigate a method of leveraging existing model checking tools to verify the partial correctness of generated code from a translator which offers no guarantees of correctness (we term such a translator an *untrusted translator*). This method operates by checking individual translations (as above) but does not require changes to the translator to do so. We investigate the feasibility of the approach through a prototype tool that verifies that Linear Temporal Logic (LTL) properties [19,20] are preserved in the translation of a Simulink model to C code.

The intuition behind the prototype is as follows: in collaboration with Rockwell Collins Inc., we have demonstrated how to verify large numbers of model requirements captured as Linear Temporal Logic (LTL) properties [19,20] over Simulink models. If we could re-verify these properties on the auto-generated C code, we would be able to provide a proof that at least the *essential properties* captured in the modeling domain are preserved in the translation from Simulink to C; the generated C code may not be fully correct with respect to the original Simulink model, but it will be *correct enough* to preserve crucial safety-critical required properties.

Our prototype tool is based on two exisiting model checkers: the NuSMV model checker [23] for the verification of Simulink models, and the ANSI-C Bounded Model Checker (CBMC) [3] for the verification of generated C code. Our tool operates by automatically translating the LTL properties expressed in the Simulink domain to monitors in the C domain, thus allowing us verify the same set of LTL properties over both the model and the generated code.

Our primary interest in this investigation was to determine if this approach would scale to address a production-sized Simulink model. Based on our experiences reported in this paper, CBMC scaled remarkably well, allowing us to easily re-verify a collection of 55 required properties on 12K+ lines of generated C code. Based on these results, we believe this approach to partial verification of auto-generated code holds great promise and we will explore the scalability further on commercial systems in future work.

The remainder of the paper is organized as follows. Section 2 contains an overview of the verification method and our proof of concept and walks through a small example. Section 3 discusses the effectiveness of our approach three

industrial examples. Section 4 compares our work with previous related work, and Section 5 concludes the paper.

2 Verification of Individual Translations

As previously mentioned, verifying properties of correctness over individual translations has been shown to be a practical method of gaining guarantees of translator correctness. Current approaches to verifying individual translations have been used successfully to verify that a variety of properties are preserved during translation, ranging from type and memory safety of compiled C programs [22] to complete equivalence of SIGNAL programs translated to C [24].

These approaches are often applied to each transformation pass performed by the translator, thus inductively proving that properties of interest are maintained during translation [25]. Most existing approaches rely on instrumenting the translator to either verify that properties are maintained during translation or provide a proof of such (which can then be verified by a proof checker). These methods are therefore of little use when an untrusted translator must be used.

In the remainder of this section, we will outline how existing model checking tools can be leveraged to provide guarantees of correctness for translations generated from untrusted translators, and describe a prototype we have developed for verifying translations from Simulink to C.

2.1 Overview of Verification

As shown in Figure 1, the general method uses three principal tools: two model checkers capable of verifying properties of interest on the input and output of the translator, respectively, and the translator itself. Two inputs are required: the translator input and a set of properties we wish to show are preserved by translation.

The method is simple. The input is first processed by the translator to produce an output. Then, the set of properties are verified on both the input and output by the appropriate model checkers, producing two lists of verified properties.

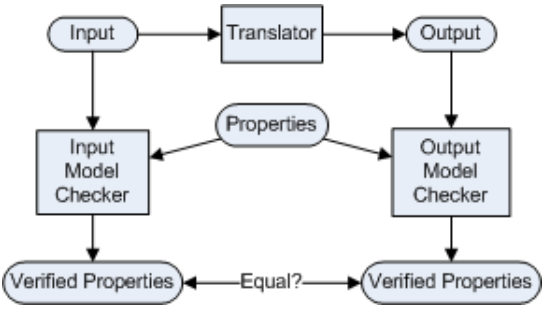


Fig. 1. General Approach to Translation Verification

These lists are compared, and any properties which fail to verify on only the input or output are reported, as they indicate that the translation either failed to preserve a property, or—in the case when a property holds on the output but not the input—removed one or more possible behaviors in the translation process. Should no such discrepancies exist, the translation has been proven to preserve the given set of properties.

Note that while we are primarily interested in this method for use in verifying the generated code, the method is applicable for any translation in which the same set of properties can be verified over both the input and output of a translator.

2.2 Prototype Implementation

Our prototype concerns the verification of C code generators for Simulink. We perform verification over sets of safety properties defined as Linear Temporal Logic (LTL) [10]. Note that a *safety property* is one that states that "something bad never happens". This is in contrast to liveness properties, which state that "something good eventually happens". We focus our efforts on safety properties for two reasons: first, it is considerably easier to verify safety properties, particularly when using bounded model checking [2]. Second, for our sample systems, all properties are expressed as safety properties, and thus the ability to verify safety properties is sufficient for our goal.

For our prototype, model checking is done by the NuSMV model checker [23] and the ANSI-C Bounded Model Checker (CBCM) [3] for Simulink [16] and C, respectively. NuSMV directly allows us to check LTL properties against finite state systems. In previous work [19,20], we developed a Simulink to NuSMV translator with Rockwell Collins Inc. This translator allows us to effectively verify the LTL properties over Simulink models, thus making NuSMV an good fit for our task.

CBMC, however, does not directly allow us to check LTL properties over C code. CBMC is designed to statically check a variety of properties such as pointer safety and safe use of arrays, as well as user-specified assertions. We therefore must express LTL properties as assertions to verify them using CBMC. We do this by translating each LTL property into a monitor (expressed as C code) that indicates violations of the LTL property using assertions.

To demonstrate the construction of a monitor, consider the following requirement from a Flight Guidance System (FGS) (this sample system is outlined in Section 3.1):

```
G((!Onside_FD_On & !Is_AP_Engaged) ->
    X(Is_AP_Engaged -> Onside_FD_On))
```

This requirement states the following: "If the onside FD cues are off, the onside FD cues shall be displayed when the AP is engaged." (In this example, the FD refers to the Flight Director – a part of the pilot's primary flight display – and the AP refers to the Auto Pilot.) Formally, the LTL formula states that it is always the case (**G**) that if the Onside FD is not on and the AP is not engaged,

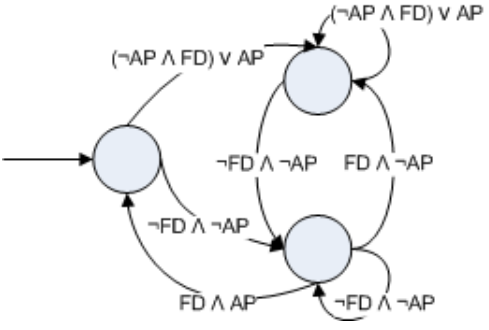


Fig. 2. LTL Property Expressed as TGBA. Note that AP represents Is_AP_Engaged and FD represents Onside_FD_On.

in the next instance in time (**X**) if the AP is engaged the Onside FD will also be on.

To convert this LTL property to a C monitor based on assertions, we first use the Spot [6] library to translate the LTL property to a Transition-based Generalized Büchi Automata (TGBA), seen in Figure 2. A TGBA is simply a Büchi automaton in which acceptance transitions are used rather than acceptance states; an input sequence is accepted if it causes an acceptance transition to be visited infinitely often [12]. Note that for this LTL property, all transitions are acceptance transitions (the reason for this is explained below).

Next, we replace the variable names taken from the LTL property with the variable names present in the generated C code (this step is specific to the naming conventions used in the translator). For this example, we use the naming conventions present in the Real-Time Workshop C code generator, available for Simulink [16]. For the example LTL property, this renaming is reflected in the C monitor described next.

Finally, we insert the C monitor into the generated C code. Note that the C code generated from our Simulink models is intended to run as a recurring task polling the environment input data. The generated C code operates by repeatedly performing three steps: poll the input, update the internal state, and produce output. LTL properties are defined over a sequence of *consistent* states; we are not interested in the system state in the middle of the next state computation, we only care when the computation has been completed. In the context of the generated C code, these consistent states occur after the output is produced and before new input is received. We therefore insert monitors between the portions of C code which produce output and receive new input.

We map the transitions defined by the TGBA to a sequence of *if* statements. Each *if* statement corresponds to a single transition in the TGBA, and the current state of the TGBA is stored as an integer. When a transition is taken, the current state of the TGBA is updated accordingly. During execution of the generated C code, a transition will be taken every "step" based upon the current values of variables in the C code and the current state of the TGBA.

```
/*LTL Formula #0
  ( G (((! Onside_FD_On) &    (! Is_AP_Engaged))
       -> ( X (Is_AP_Engaged -> Onside_FD_On)))) */
 if ((ltlstate0 == 1) && ((!FGS_Y.Is_AP_Engaged && !FGS_Y.Onside_FD_On))) {
      ltlstate0 = 3;
 } else if ((ltlstate0 == 1) && ((!FGS_Y.Is_AP_Engaged
    && FGS_Y.Onside_FD_On) || (FGS_Y.Is_AP_Engaged))) {
      ltlstate0 = 2;
 } else if ((ltlstate0 == 2) && (!FGS_Y.Is_AP_Engaged && !FGS_Y.Onside_FD_On)) {
      ltlstate0 = 3;
 } else if ((ltlstate0 == 2) && ((!FGS_Y.Is_AP_Engaged
    && FGS_Y.Onside_FD_On) || (FGS_Y.Is_AP_Engaged))) {
      ltlstate0 = 2;
 } else if ((ltlstate0 == 3) && (!FGS_Y.Is_AP_Engaged && !FGS_Y.Onside_FD_On)) {
      ltlstate0 = 3;
 } else if ((ltlstate0 == 3) && (!FGS_Y.Is_AP_Engaged && FGS_Y.Onside_FD_On)) {
      ltlstate0 = 2;
 } else if ((ltlstate0 == 3) && (FGS_Y.Is_AP_Engaged && FGS_Y.Onside_FD_On)) {
      ltlstate0 = 1;
 } else { assert(0); }
```

Fig. 3. LTL Property Expressed as C Assertion

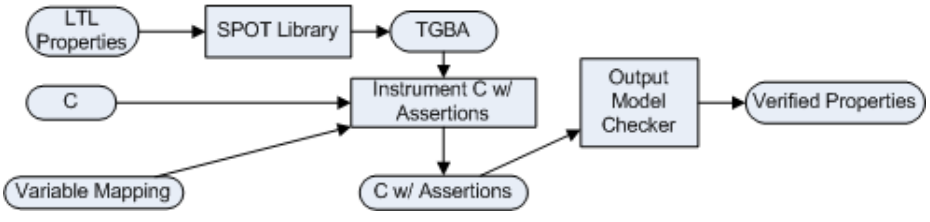


Fig. 4. Verifying LTL Properties on C Code

If no transition can be taken (indicating an input sequence not accepted by the TGBA), an error is signaled by means of an $assert(0)$ statement. The monitor corresponding to our example property is seen in Figure 3.

Once monitors are created, they are inserted into the C code we wish to verify. CBMC can then statically determine if there exists an execution path which reaches the monitor's assertion, thus checking if the LTL property holds over the C code. An outline of this entire process as performed by our prototype is shown in Figure 4.

Our prototype contains several details worth noting. As mentioned above, to keep our monitor construction simple, we have elected to only verify safety properties. Formally, these are expressed as $\mathbf{G}(p)$, where p is some property which does not use the finally ($\mathbf{F}$) operator. ($\mathbf{F}$ is used to define liveness properties, which we do not support.) For a TGBA corresponding to a safety property, all transitions are accepting transitions, and, thus, any input sequence which causes the TGBA to always correctly transition is accepted. This property is key to our monitor's correctness—if we attempted to capture liveness properties, we might fail to reject an input sequence which always correctly transitions but does not infinitely pass by an accepting transition. Note that it is possible to express liveness properties as safety properties [1], thus allowing us to extend monitor

construction to handle liveness properties, though this would require significantly more development effort of our prototype tool. Additionally, expressing liveness properties as safety properties requires the introduction of state recording functionality to the system, likely increasing the size of the searchable state space by a significant amount.

Second, our prototype contains several aspects which are unproven, but must be assumed to work if a user of the prototype wishes to claim a partial proof of correctness. Specifically, errorneous behavior in the variable mapping, monitor creation, or model checkers could lead to either false postives (signaling a property violation in correctly generated code) or false negatives (failing to signal a property violation in incorrectly generated code). However, with the notable exception of the model checkers, the prototype is likely much simpler than the code generator. We therefore believe trusting these prototype components is reasonable.

Finally, note that unlike NuSMV, CBMC is a bounded model checker [3]. Bounded model checkers do not provide a complete proof of correctness for a property unless the search depth of the model checker exceeds the *completeness threshold* [2]. In other words, for a property to be proven to hold on a system with a bounded model checker, a sufficient number of states must have been explored by the model checker. For safety properties that do not use temporal operators besides the initial Globally (i.e., $\mathbf{G}(p)$ where p contains no temporal operators), this threshold is the *reachability diameter* of the model [2]. Using the symbolic model checker in NuSMV and the Simulink model, we can calculate the reachability diameter and use the result to achieve complete verification when using CBMC. Note that it is possible for an incorrect code generator to increase the reachability diameter (though plausible occurrences of this are difficult to formulate); to be conservative, we therefore used a search depth twice the reachability diameter of the Simulink model when verifying properties on the generated C code using CBMC.

For safety properties in which p contains the next operator ($\mathbf{X}$), we use an approach suggested in [2]. Using this approach, we extend the original model with an automaton derived from p (using essentially the same approach used to create C monitors) before calculating the reachability diameter, thus accounting for the state tracking variables introduced by the creation of the C monitors described above.

3 Application Results

We applied our prototype to three sample systems (two small toy-examples and one close to production model of the model-logic of a transport class flight guidance system) using two different code generators: Real-Time Workshop, a commercial code generator from Mathworks [16], and a currently in-development code generator courtesy of Rockwell Collins Inc. Our results showed our prototype scaled remarkably well, managing to verify 55 LTL properties over the translation of a Simulink model of significant size.

Initially, we verified single LTL properties over small (but realistic) sample systems. These initial attempts were successful, showing the LTL properties held over the translations in question, and ran very quickly. We then explored the scalability and robustness of our prototype by successfully verifying the translation of a much larger model using a set of 55 LTL properties. These properties used a number of different operators (e.g., the next operator, equivalence operator, implies operator, etc.) and were structured in a variety of ways. Additionally, these properties were defined over a large portion of the input and output variables. We thus feel that these properties give a good indication of the correctness of a translation from Simulink to C, the robustness of our prototype, and the feasibility of the method in general. The models and results are described in more detail below.

3.1 Sample Systems

The three sample systems are described below. Measurements relating the size of the systems are given in Table 1. Generated C code lines of code (LOC) counts were performed by SLOCCount [27].

Altitude Switch (ASW): The Altitude Switch (ASW) is a re-useable component that turns power on to a Device Of Interest (DOI) when the aircraft descends below a threshold altitude above ground level (AGL). If the altitude cannot be determined for more than two seconds, the ASW indicates a fault. The detection of a fault turns on an indicator lamp within the cockpit.

Wheel Brake System (WBS): The Wheel Brake System (WBS) is a Simulink model derived from the WBS case example found in ARP 4761 [26,15]. The WBS is installed on the two main landing gears. Braking on the main gear wheels is used to provide safe retardation of the aircraft during the taxiing and landing phases, and in the event of a rejected take-off. Braking on the ground is either commanded manually, via brake pedals, or automatically (autobrake) without the need for pedal application. The Autobrake function allows the pilot to pre-arm the deceleration rate prior to takeoff or landing. When the wheels have traction, the autobrake function will control break pressure to provide a smooth and constant deceleration.

Flight Guidance System (FGS): A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. It compares

Table 1. Measurements of Sample System Size. The columns labeled RTW and In-Dev refer to the results from Real Time Workshop and the Rockwell Collins code generators respectively.

	Simulink Nodes	**System Diameter**	**RTW C LOC**	**In-Dev C LOC**
ASW	14	3	220	134
WBS	157	2	774	197
FGS	4510	10	12,242	1,379

Table 2. Time to Verify SMV and Generated C Code for our sample systems

	Real-Time Workshop C	In-Development C	SMV
ASW	7 secs.	8 secs.	6 secs.
WBS	57 secs.	9 secs.	2 secs.
FGS	1002 secs.	273 secs.	66 secs.

the measured state of an aircraft (position, speed, and altitude) to the desired state and generates pitch and roll-guidance commands to minimize the difference between the measured and desired state. The FGS consists of the mode logic, which determines which lateral and vertical modes of operation are active and armed at any given time, and the flight control laws that accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. In this study we have used the model of the mode logic.

3.2 Results

For the C code generations performed by both code generators, all LTL properties were verified for each case example. The time to verify properties over the input and output aspects of translations is given in Table 2. All properties were verified on a Intel Centrino Duo machine running at 1.83 GHz with 1 GB RAM.

We feel the FGS is the most interesting case example, being both the largest model and the model with the largest number of associated LTL properties. Translation of the FGS Simulink model yielded over 12,000 lines of source-code with Real Time Workshop and over 1,300 lines of code with the Collins In-Development generator (Table 1). Verification of the LTL properties over the generated C code took approximately 17 and 5 minutes respectively (Table 2). As a reference, verification of the properties over the NuSMV model extracted from the Simulink model took approximately 1 minute. Given the the results for a substantial system such as the FGS (in terms of code size as well as number of properties to verify), we feel that our prototype demonstrates that the method of partial correctness verification of auto-generated C-code using model-checking techniques is both feasible and useful.

4 Related Work

The concept of a verifying translator originated in early work by Floyd [9]. This was followed by a number of attempts centered on proving that a translator would produce correct output for any input [7,13,21]. While these attempts met with some success, manual proofs are difficult to perform and automatic proofs of translator correctness remain generally out of reach.

More recently, several efforts have focused on proving properties for individual translations correct, rather than proving the translator correct in general. Necula and Lee have developed a compiler for a type-safe subset of C that verifies type and memory safety of the resulting assembler using code annotations [22]. They

later extended this idea to the full Java language [4]. Rinard has put forth the logical foundations for "credible compilation" of imperative languages in which a proof is generated showing the equivalence of the compiler's input and output [25]. Pnueli et. al. describe a similar approach to Rinard's and illustrate their approach using a C code generator for the synchronous language SIGNAL [24]. Note that both Pnueli's and Rinard's approaches generate proofs of equivalence between the input and output, while Necula and Lee's approach only produces a proof that the output satisfies certain properties which are known to exist in the input (such as type safety).

These recent efforts share our goal of verifying properties of individual translation, but differ from the approach described in this report in that they are implemented within the translator, and, thus, must be incorporated by the developers of the translator. In contrast, the approach we have presented can be applied to any translator, provided a method of statically verifying a set of properties over both the translator input and output exists (in the case of our prototype, this meant a model checker for both the input and output and knowledge of how to create and insert monitors into the generated C code). Additionally, while these approaches are clearly more feasible than proving a translator correct in general, they do require significant effort on the part of the translator developer, the approach we have presented requires no changes to the translator itself.

Denney and Fischer have developed a method of automatically annotating generated code without modifying the translator [5]. This method achieves similar goals in that individual code generations can be statically verified without the need to instrument the code generator itself. However, method of static verification used differs - rather than directly verifying a set properties are true for generated code, Denney and Fischer's method instead opts to annotate the generated code such that the correctness of the annotations implies the correctness of one or more safety properties. Once the code is annotated, a proof checker is used to verify the annotations.

The construction of C monitors in our prototype is performed similarly in a tool developed by Giannakopoulou et al. [11], though their work focuses on the generation of monitors for run-time verification.

5 Conclusions

In this paper, we have explored a method by which useful properties can be verified to hold over individual translations performed by untrusted translators. We have shown that though widespread adoption of translators supporting translation validation has not yet been realized, and provably correct translators still remain difficult to build, the use of already existing model checkers provides a feasible means of establishing guarantees of translation correctness beyond the guarantees generally available.

In particular, we show the applicability of this method in the context of code generators used in model-based development, and have developed a prototype leveraging the NuSMV and CBMC model checkers. These model checkers, in

conjunction with the infrastructure to translate LTL properties into C assertions, are used to verify that a set of LTL properties are preserved when generating C code from Simulink models. Notably, we applied our prototype to demonstrate that two code generators generated C code for a large industrial Simulink model correctly with respect to a set of 55 LTL properties. We believe this demonstrates both scalability and effectiveness of this method and intend to explore this method further in the near future.

We feel that this approach holds promise as a relatively easy to implement technique that can be employed by software developers concerned with the correctness of a untrusted translator, specifically code generators. This approach allows developers without access to the internals of a code generator (or without the desire to implement translation validation methods into a code generator) to achieve some guarantee of the correctness of an individual translation.

Acknowledgements

We would like to thank Dr. Michael Whalen from Rockwell Collins Inc. for allowing us to use his in-development C code generator, as well as for insights and discussions on the use of bounded model checkers.

References

1. Biere, A., Artho, C., Schuppan, V.: Liveness Checking as Safety Checking. Electronic Notes in Theoretical Computer Science 66(2), 160–177 (2002)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers 58, 118–149 (2003)
3. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988. Springer, Heidelberg (2004)
4. Colby, C., Lee, P., Necula, G.C., Blau, F., Plesko, M., Cline, K.: A certifying compiler for Java. ACM SIGPLAN Notices 35(5), 95–107 (2000)
5. Denney, E., Fischer, B.: Annotation inference for the safety certification of automatically generated code. In: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE 2006), pp. 265–268 (2006)
6. Duret-Lutz, A., Poitrenaud, D.: SPOT: an extensible model checking library using transition-based generalized Bu/spl uml/chi automata. In: Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004 (MASCOTS 2004), pp. 76–83 (2004)
7. Dybjer, P.: Using Domain Algebras to Prove the Correctness of a Compiler. Springer, Heidelberg
8. Esterel-Technologies. SCADE Suite product description (2004), http://www.esterel-technologies.com/v2/scadeSuiteForSafetyCriticalSoftwareDevelopment/index.html
9. Floyd, R.W.: Assigning meanings to programs. Mathematical Aspects of Computer Science 19(19-32), 1 (1967)

10. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Proceedings of the Fifteenth IFIP WG6.1 International Symposium on Protocol Specification, Testing and Verification XV, pp. 3–18. Chapman & Hall, Ltd., Boca Raton (1996)
11. Giannakopoulou, D., Havelund, K.: Automata-Based Verification of Temporal Properties on Running Programs. In: Proceedings of International Conference on Automated Software Engineering (ASE 2001), pp. 412–416 (2001)
12. Giannakopoulou, D., Lerda, F.: From states to transitions: Improving translation of LTL formulae to Buchi automata. In: Proceedings of the 22nd IFIP WG, pp. 6 (2002)
13. Guttman, J.D., Ramsdell, J.D., Wand, M.: VLISP: A verified implementation of Scheme. Higher-Order and Symbolic Computation 8(1), 5–32 (1995)
14. Hoare, T.: The Verifying Compiler: A Grand Challenge for Computing Research. In: Böszörményi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789. Springer, Heidelberg (2003)
15. Joshi, A., Heimdahl, M.P.E.: Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In: Winther, R., Gran, B.A., Dahll, G. (eds.) SAFECOMP 2005. LNCS, vol. 3688, pp. 122–135. Springer, Heidelberg (2005)
16. Mathworks Inc. Simulink product web site, http://www.mathworks.com/products/simulink
17. Mathworks Inc. Stateflow product web site, http://www.mathworks.com
18. McCarthy, J.: Towards a mathematical science of computation. Information Processing 62, 21–28 (1962)
19. Miller, S.P., Anderson, E.A., Wagner, L.G., Whalen, M.W., Heimdahl, M.P.E.: Formal Verification of Flight Critical Software. In: Proceedings of the AIAA Guidance, Navigation and Control Conference and Exhibit (August 2005)
20. Miller, S.P., Tribble, A.C., Whalen, M.W., Heimdahl, M.P.E.: Proving the shalls: Early validation of requirements through formal methods. Int. J. Softw. Tools Technol. Transf. 8(4), 303–319 (2006)
21. Moore, J.S.: A mechanically verified language implementation. Journal of Automated Reasoning 5(4), 461–492 (1989)
22. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. ACM SIGPLAN Notices 33(5), 333–344 (1998)
23. The NuSMV Toolset (2005), http://nusmv.irst.itc.it/
24. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS, vol. 98, pp. 151–166
25. Rinard, M.: Credible compilation. In: Proceedings of the FLoC Workshop Run-Time Result Verification (July 1999)
26. ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. SAE International (December 1996)
27. Wheeler, D.: SLOCCount: Source Lines of Code Count. Webpage. Version, 2 (2004)

A Practical Approach to Partiality – A Proof Based Approach*

Farhad Mehta

Systransis AG – Transport Information Systems,
Bahnhofplatz, P.O. Box 4714,
CH-6304 Zug, Switzerland

Abstract. Partial functions are frequently used when specifying and reasoning about computer programs. Using partial functions entails reasoning about potentially ill-defined expressions. In this paper we show how to formally reason about partial functions without abandoning the well understood domain of classical two-valued predicate calculus. In order to achieve this, we *extend* standard predicate calculus with the notion of *well-definedness* which is currently used to *filter out* potentially ill-defined statements from proofs. The main contribution of this paper is to show how the standard predicate calculus can be extended with a new set of *derived* proof rules that can be used to *preserve* well-definedness in order to make proofs involving partial functions less tedious to perform.

1 Introduction

Partial functions are frequently used when specifying and reasoning about computer programs. Some basic mathematical operations (such as division) are partial, some basic programming operations (such as array look-ups or pointer dereferencing) are partial, and many functions that arise through recursive definitions are partial or possibly non-terminating . Using partial functions entails reasoning about potentially ill-defined expressions (such as $3/0$) in proofs which (as discussed later in §3 and §4) can be tedious and problematic to work with. Providing proper logical and tool support for reasoning in the presence of partial functions is therefore important in the engineering setting. Although the contributions of this paper are theoretical in nature, they result in practical benefits which will be stated later in this section.

The current approaches for explicitly reasoning in the partial setting [6,7,19] are based on three-valued logic where the *valuation* of a predicate is either true, false, or undefined (for predicates containing ill-defined expressions). They also propose their own 'special-purpose' proof calculi for performing such proofs. Using such a special-purpose proof calculus has the drawback that it differs from the standard predicate calculus. For instance, it may disallow the use of the

* This research was carried out at the ETH Zurich as part of the EU research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 238–257, 2008.

law of excluded middle (to avoid proving '$3/0 = x \vee 3/0 \neq x$'). Such differences make any special-purpose calculus unintuitive to use for someone well versed in standard predicate calculus since, for instance, it is hard to mentally anticipate the consequences of disallowing the law of excluded middle on the validity of a logical statement. Automation too requires additional effort since the well-developed automated theorem proving support already present for standard predicate calculus [22] cannot be readily reused. Additionally, as stated in [8], there is currently no consensus on which is the 'right' calculus to use.

In this paper we present a general methodology for using standard predicate calculus to reason in the 'partial' setting by *extending* it with new syntax and derived rules. We then *derive* one such special-purpose calculus using this general methodology. We call our approach 'proof based' since (in the spirit of [4] that we build on) we do not make any detours through three-valued semantic arguments (which can be found in [8] and [7]), but confine our reasoning to (syntactic) proofs in standard predicate calculus. The novelty of this approach is that we are able to reduce *all* our reasoning (i.e. within our system, as well as about it) to standard predicate calculus, which is both widely understood and has well-developed automated tool support. This approach additionally gives us a theoretical basis for comparing the different special-purpose proof calculi already present(which is done in §8.1), and the practical benefit of being able to exchange proofs and theorems between different theorem proving systems.

The ideas presented in this paper have additionally resulted in providing better tool support for theorem proving in the partial setting within the RODIN development environment [1] for Event-B [3].

Setting. Our practical setting is that of formal system development in Event-B [3]. The development process consists of *modeling* a desired system and *proving* proof obligations arising from it. The logic used in Event-B is set theory built on first-order predicate calculus. A user can define partial functions in this logic. The results presented here are independent of the Event-B method and its set theory. They are equally applicable in many areas where predicate calculus is used to reason in a setting with potentially ill-defined expressions.

Structure. In §2 we define the syntax (§2.1) and proof rules (§2.2) for standard first-order classical predicate calculus with equality (*FoPCe*). In §2.3 we state the forms of reasoning we use in this paper and in §3 we show how partial functions are defined. In §4 we show how to separate the concerns of well-definedness from those of validity by filtering out ill-defined proof obligations using the well-definedness (WD) operator '$\mathcal{D}$' as in [4,7,8]. In §5 we describe $\mathcal{D}$ and state some of its important properties. The *main contribution* of this paper is in §6 where we show how the notion of well-definedness can be integrated into standard predicate calculus. In §6.1 we extend the definition of $\mathcal{D}$ to sequents. We then formally define the notions of a well-defined sequent (§6.2) and a well-definedness (WD) preserving proof rule (§6.3). In §6.4 we derive a proof calculus ($FoPCe_{\mathcal{D}}$) that preserves well-definedness. In §7 we return to the practical issue of filtering and proving proof obligations. In §8 we compare our approach with related work

and show how our approach can be used as a basis to compare the special-purpose proof calculi presented in [6] and [7]. We conclude in §9 by stating what we have achieved and its impact on the RODIN development environment [1] for Event-B.

2 Predicate Calculus

In this section we define the syntax and proof rules for the standard (first-order, classical) predicate calculus with equality that we will use in the rest of the paper.

2.1 Basic Syntax

Basic formulæ Formulæ in first-order predicate calculus can either be predicates (P) or expressions (E). We define the structure of *basic formulæ* as follows:

$$\begin{array}{lcl} P & := & \bot \mid \neg P \mid P \wedge P \mid \forall x.P \mid E = E \mid R(\vec{E}) \\ E & := & x \mid f(\vec{E}) \end{array}$$

Where $\bot$ is the 'false' predicate, x is a variable, $\vec{E}$ is a finite, ordered sequence of expressions, R is a relational predicate symbol, and f is a function symbol. Equality is denoted by the infix binary relational predicate symbol '='.

Sequents. A sequent is a statement we want to prove, denoted '$\mathsf{H} \vdash G$', where H is a finite set of predicates (the hypotheses), and G is a single predicate (the goal).

We extend this basic syntax in §2.2, §5, and §6 using syntactic definitions such as '$\top \mathrel{\widehat{=}} \neg\bot$'. The '$\widehat{=}$' symbol represents *syntactic* equivalence. It is not itself part of the syntax, but a *meta-logical* connective.

2.2 Proof Rules for *FoPCe*

Here are the rule schemas that define the basic proof calculus for first-order predicate calculus with equality (*basicFoPCe*):

$$\frac{}{\mathsf{H}, P \vdash P}\ hyp \qquad \frac{\mathsf{H} \vdash Q}{\mathsf{H}, P \vdash Q}\ mon \qquad \frac{\mathsf{H} \vdash P \quad \mathsf{H}, P \vdash Q}{\mathsf{H} \vdash Q}\ cut \qquad \frac{\mathsf{H}, \neg P \vdash \bot}{\mathsf{H} \vdash P}\ contr$$

$$\frac{}{\mathsf{H}, \bot \vdash P}\ \bot hyp \qquad \frac{\mathsf{H}, P \vdash \bot}{\mathsf{H} \vdash \neg P}\ \neg goal \qquad \frac{\mathsf{H} \vdash P}{\mathsf{H}, \neg P \vdash Q}\ \neg hyp$$

$$\frac{\mathsf{H} \vdash P \quad \mathsf{H} \vdash Q}{\mathsf{H} \vdash P \wedge Q}\ \wedge goal \qquad \frac{\mathsf{H}, P, Q \vdash R}{\mathsf{H}, P \wedge Q \vdash R}\ \wedge hyp$$

$$\frac{\mathsf{H} \vdash P}{\mathsf{H} \vdash \forall x \cdot P}\ \forall goal\ (x \ \underline{\mathsf{nfin}}\ \mathsf{H}) \qquad \frac{\mathsf{H}, [x := E]P \vdash Q}{\mathsf{H}, \forall x \cdot P \vdash Q}\ \forall hyp$$

$$\frac{}{\mathsf{H} \vdash E = E}\ = goal \qquad \frac{\mathsf{H} \vdash [x := E]P}{\mathsf{H}, E = F \vdash [x := F]P}\ = hyp$$

Syntactic operators. The rules shown above contain occurrences of so-called *syntactic operators* for substitution and non-freeness. The predicate '$[x := E]P$' denotes the syntactic operator for substitution $[x := E]$, applied to the predicate P. The resulting predicate is P, with all free occurrences of the variable x replaced by the expression E. The side condition '$(x \ \mathsf{nfin}\ \mathsf{H})$' asserts that the variable 'x' is not free in any of the predicates contained in 'H'. Both these syntactic operators are defined (using '$\widehat{=}$') on the inductive structure of basic formulæ in such a way that they can always be evaluated away. Their formal definitions can be found in [2]. Syntactic operators can be thought of as 'macros' whose repeated replacement always results in a basic formula. Our basic syntax for formulæ therefore does not need to be extended to take the syntactic operators into account.

Derived Logical Operators. The other standard logical operators $\top$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\exists$ can be expressed in terms of the above basic logical connectives using syntactic definitions such as '$\top \mathrel{\widehat{=}} \neg\bot$'. Their corresponding proof rules can then be derived from the above basic proof rules. Both these steps are standard and their details can be found in §3.3 of [17].The resulting syntax and proof rules correspond to the standard first-order predicate calculus with equality. We will refer to this collection of proof rules as the theory (or proof calculus) *FoPCe*.

2.3 Reasoning

There are two forms of reasoning that we use in this paper. The first is *syntactic rewriting* using syntactic definitions such as '$\top \mathrel{\widehat{=}} \neg\bot$', where all occurrences of '$\top$' in a formula may be replaced with '$\neg\bot$', purely on the syntactic level, to get a syntactically equivalent formula. The second is *logical validity* where we additionally appeal to the notion of proof. All proofs done in this paper use the standard predicate calculus *FoPCe*. When we say that a predicate 'P' is provable, we mean that we have a proof of the sequent '$\vdash P$' using *FoPCe*. Most proofs done in this paper rely on both syntactic and logical reasoning. In §5.2 we discuss precautions we need to take when performing such proofs.

3 Defining Partial Functions

In this section we show how a partial function is defined in our mathematical logic. We first present how this can be done in general, and then follow with an example that will be used in the rest of this paper.

3.1 Conditional Definitions

A partial function symbol 'f' is defined using the following *conditional definition*:

$$\frac{}{C^f_{\vec{x}} \ \vdash\ y = f(\vec{x}) \ \Leftrightarrow\ D^f_{\vec{x},y}}\ f_{def}$$

Conditional definitions of the above form can safely be added to *FoPCe* as an axiom, provided:

1. The variable 'y' is not free in the predicate '$C^f_{\vec{x}}$'.
2. The predicate '$D^f_{\vec{x},y}$' only contains the free variables from '$\vec{x}$' and 'y'.
3. The predicates '$C^f_{\vec{x}}$' and '$D^f_{\vec{x},y}$' only contain previously defined symbols.
4. The theorems:
   Uniqueness: $C^f_{\vec{x}} \vdash \forall y, z \cdot\ D^f_{\vec{x},y} \wedge D^f_{\vec{x},z} \ \Rightarrow\ y = z$
   Existence: $C^f_{\vec{x}} \vdash \exists y \cdot\ D^f_{\vec{x},y}$
   must both be provable using *FoPCe* and the previously introduced definitions.

The predicate '$C^f_{\vec{x}}$' is the *well-definedness condition* for 'f' and specifies its domain. For a total function symbol, '$C^f_{\vec{x}}$' is '$\top$'. Provided '$C^f_{\vec{x}}$' holds, f_{def} can be used to eliminate all occurrences of 'f' in a formula in favor of its definition '$D^f_{\vec{x},y}$'. More details on conditional definitions can be found in [4].

3.2 Recursive Definitions

Note that conditional definitions as described above cannot be directly used to define function symbols recursively since the definition of a function symbol '$D^f_{\vec{x},y}$' may not itself contain the function symbol 'f' that it defines, as stated in the third condition above.

It is still possible to define partial functions recursively in a theory (such as the set theory described in [4] and [3]) which supports the applications of functions that are *expressions* (i.e. not plain function symbols) in the theory. Such recursively defined functions are then defined as *constant* symbols (i.e. total function symbols with no parameters). Function application is done using an additional function symbol for function application (often denoted using the standard function application syntax '$\cdot(\cdot)$') with two parameters which are both expressions: the function to apply, and the expression to apply it to. The definition of this function application symbol is conditional. Its well definedness predicate ensures that the its first parameter is indeed a function, and its second parameter is an expression that belongs to the domain of this function. This methodology is described in detail in [4] which also describes (in §1.5) how functions can be defined recursively.

3.3 A Running Example

For our running example, let us assume that our syntax contains the nullary function symbol '0', and the unary function symbol '*succ*', and our theory contains the rules for Peano arithmetic. We may now introduce a new unary function symbol '*pred*' in terms of '*succ*' using the following conditional definition:

$$\frac{}{E \neq 0 \ \vdash\ y = pred(E) \ \Leftrightarrow\ succ(y) = E}\ pred_{def}$$

Defined in this way, '*pred*' is partial since its definition can only be unfolded when we know that its argument is not equal to 0. The expression '$pred(0)$' is still a syntactically valid expression, but is *under-specified* since we have no way

of unfolding its definition. The expression '$pred(0)$' is therefore said to be *ill-defined*. We do not have any way to prove or refute the predicate '$pred(0) = x$'.

The predicates '$pred(0) = pred(0)$' and '$pred(0) = x \vee pred(0) \neq x$' though, can still be proved to be valid in *FoPCe* on the basis of their logical structure. This puts us in a difficult position since these predicates contain ill-defined expressions. The standard proof calculus *FoPCe* is therefore not suitable if we want to restrict our notion of validity only to sequents that do not contain ill-defined formulæ.

4 Separating WD and Validity

Since our aim is to still be able to use *FoPCe* in our proofs, we are not free to change our notion of validity. We instead take the pragmatic approach of *separating* the concern of validity from that of well-definedness and require that both properties hold if we want to avoid proving potentially ill-defined proof obligations. In this case, we still allow the predicate '$pred(0) = pred(0)$' to be proved to be valid, but we additionally ensure that it cannot be proved to be well-defined. When proving a proof obligation '$\mathsf{H} \vdash G$' we are then obliged to prove two proof obligations:

$\mathsf{WD}: \quad \vdash \mathcal{D}(\mathsf{H} \vdash G)$	$\mathsf{Validity}: \quad \mathsf{H} \vdash G$

The first proof obligation, WD, is the well-definedness proof obligation for the sequent '$\mathsf{H} \vdash G$'. It is expressed using the well-definedness (WD) operator $\mathcal{D}$ that is introduced in §5 and defined for sequents in §6.1. The second proof obligation, Validity, is its validity proof obligation. An important point to note here is that *both these proof obligations may be proved using FoPCe*.

Proving WD can be seen as *filtering out* proof obligations containing ill-defined expressions. For instance, for '$\vdash pred(0) = pred(0)$' we are additionally required to prove '$\vdash 0 \neq 0 \wedge 0 \neq 0$' as its WD (this proof obligation is computed using definitions that appear in §5 and §6.1). Since this is not provable, we have filtered out (and therefore rejected) '$\vdash pred(0) = pred(0)$' as not being well-defined in the same way as we would have filtered out and rejected '$\vdash 0 = \varnothing$' as not being well-typed. Well-definedness though, is undecidable and therefore needs to be proved. Figure 1 illustrates how well-definedness can be thought of as an additional *proof-based* filter for mathematical texts.

When proving Validity, we may then additionally assume that the initial sequent '$\mathsf{H} \vdash G$' is well-defined. The assumption that a sequent is well-defined can

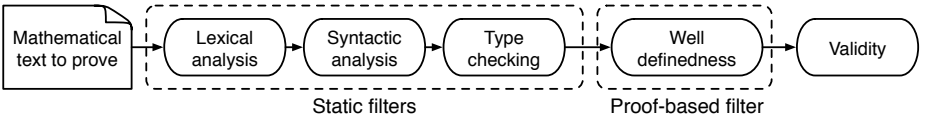


Fig. 1. Well-definedness as an additional filter

be used to greatly ease its proof. It allows us to avoid proving that a formula is well-defined every time we want to use it (by expanding its definition, or applying its derived rules) in our proof. For instance we may apply the simplification rule '$x \neq 0 \vdash pred(x + y) = pred(x) + y$' without proving its premise '$x \neq 0$'. This corresponds to the way a mathematician works.

In §6.4 we show this key result formally; i.e. how *conditional definitions become 'unconditional'* for well-defined sequents.

For the moment though, we may only assume that the initial sequent of Validity is well-defined. In order to take advantage of this property throughout a proof we need to use proof rules that *preserve* well-definedness. Preserving well-definedness in an interactive proof also has the advantage of preventing the user from introducing possibly erroneous ill-defined terms into a proof. A proof calculus preserving well-definedness is presented in §6. Before that, in the next section, we first describe the well-definedness operator.

5 The Well-Definedness Operator

The WD operator '$\mathcal{D}$' formally encodes what we mean by well-definedness. $\mathcal{D}$ is a syntactic operator (similar in status to the substitution operator '$[x := E]$' seen in §2.2) that maps formulæ to their well-definedness (WD) predicates. We interpret the predicate denoted by $\mathcal{D}(F)$ as being valid iff F is well-defined. The $\mathcal{D}$ operator is attributed to Kleene [16] and also appears in [4,7,8], and as the 'δ' operator in [6,11].

Since $\mathcal{D}$ has been previously well studied, we only give in this section an overview of the properties of $\mathcal{D}$ from [4] that we use later in this paper. In §5.1 we define $\mathcal{D}$ for formulæ in *basicFoPCe*. In §5.2 we derive equivalences that allow $\mathcal{D}$ for all formulæ in *FoPCe* to be computed, and state an important properties of $\mathcal{D}$ that we use later in §6.

5.1 Defining $\mathcal{D}$

$\mathcal{D}$ is defined on the structure of formulæ in *basicFoPCe* using syntactic definitions. For expressions, $\mathcal{D}$ is defined as follows:

$$\mathcal{D}(x) \mathrel{\widehat{=}} \top \tag{1}$$

$$\mathcal{D}(f(\vec{E})) \mathrel{\widehat{=}} \vec{\mathcal{D}}(\vec{E}) \wedge C^f_{\vec{E}} \tag{2}$$

where $\vec{\mathcal{D}}$ is $\mathcal{D}$ extended for sequences of formulæ (i.e. $\vec{\mathcal{D}}(\,) \mathrel{\widehat{=}} \top$, $\vec{\mathcal{D}}(F, \vec{F}) \mathrel{\widehat{=}} \mathcal{D}(F) \wedge \vec{\mathcal{D}}(\vec{F})$). An occurrence of a variable in a formula is always well-defined. The occurrence of a function application is well-defined iff all its operands are well-defined (i.e. $\vec{\mathcal{D}}(\vec{E})$ holds), and the well-definedness condition '$C^f_{\vec{E}}$' from the conditional definition of f holds. The resulting definition for the running example '*pred*' is $\mathcal{D}(pred(E)) \mathrel{\widehat{=}} \mathcal{D}(E) \wedge E \neq 0$.

Similarly, $\mathcal{D}$ for $\bot$, $\neg$, $=$ and relational predicate application is defined as follows:

$$\mathcal{D}(\bot) \mathrel{\widehat{=}} \top \tag{3}$$
$$\mathcal{D}(\neg P) \mathrel{\widehat{=}} \mathcal{D}(P) \tag{4}$$
$$\mathcal{D}(E_1 = E_2) \mathrel{\widehat{=}} \mathcal{D}(E_1) \wedge \mathcal{D}(E_2) \tag{5}$$
$$\mathcal{D}(R(\vec{E})) \mathrel{\widehat{=}} \vec{\mathcal{D}}(\vec{E}) \tag{6}$$

Note that we regard relational predicate application as always being total. In case we require partial relational predicate symbols, they can be supported in the same way as partial function symbols.

Since we would like predicates such as '$x \neq 0 \wedge \mathit{pred}(x) = x$' (or similarly, '$x \neq 0 \Rightarrow \mathit{pred}(x) \neq x$') to be well-defined special care is taken while defining the well-definedness of $\wedge$ and $\forall$ as follows:

$$\mathcal{D}(P \wedge Q) \mathrel{\widehat{=}} (\mathcal{D}(P) \wedge \mathcal{D}(Q)) \vee (\mathcal{D}(P) \wedge \neg P) \vee (\mathcal{D}(Q) \wedge \neg Q) \tag{7}$$
$$\mathcal{D}(\forall x \cdot P) \mathrel{\widehat{=}} (\forall x \cdot \mathcal{D}(P)) \vee (\exists x \cdot \mathcal{D}(P) \wedge \neg P) \tag{8}$$

Intuitively, the above definitions enumerate *all* the possible conditions where a conjunctive or universally quantified predicate *could* be well-defined. From these definitions we can see that $\mathcal{D}$ is itself total and can always be eliminated from any formula.

A formal derivation of the above definitions can be found in [4] and a semantic treatment of $\mathcal{D}$ can be found in [8], [7], and [10], but for the purpose of this paper it is sufficient to accept the above syntactic equivalences as the definition of $\mathcal{D}$.

5.2 Proving Properties about $\mathcal{D}$

In this section (and in §6.4) we show some important *logical* (as opposed to syntactic) properties about $\mathcal{D}$. Care must be taken when proving statements that contain both syntactic and logical operators. Since $\mathcal{D}$ is not a logical operator, but a syntactic one, modifying its argument using standard logical transformations is not valid. For instance, given that '$P \Leftrightarrow Q$' holds (i.e. is valid in *FoPCe*), it is wrong to conclude that '$\mathcal{D}(P) \Leftrightarrow \mathcal{D}(Q)$' (consider the valid predicate '$\top \Leftrightarrow \mathit{pred}(0) = \mathit{pred}(0)$'). The only modifications that can be made to the arguments of $\mathcal{D}$ are purely syntactic ones, such as applying syntactic rewrites (using syntactic definitions that use '$\widehat{=}$'). In this section we state some properties of $\mathcal{D}$.

$\mathcal{D}$ of WD predicates. An important property of $\mathcal{D}$ is that for any formula F,

$$\mathcal{D}(\mathcal{D}(F)) \Leftrightarrow \top \tag{9}$$

This means that all WD predicates are themselves well-defined . This property can be proved by induction on the structure of basic formulæ. A proof of this nature can be found in the appendix of [4]. Note that the above property is expressed in terms of logical equivalence '$\Leftrightarrow$' and not syntactic definition '$\widehat{=}$'.

$\mathcal{D}$ for Derived Logical Operators. The following equivalences can be used to compute the WD predicates of the derived logical operators $\top$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\exists$:

$$\mathcal{D}(\top) \Leftrightarrow \top \tag{10}$$
$$\mathcal{D}(P \vee Q) \Leftrightarrow (\mathcal{D}(P) \wedge \mathcal{D}(Q)) \vee (\mathcal{D}(P) \wedge P) \vee (\mathcal{D}(Q) \wedge Q) \tag{11}$$
$$\mathcal{D}(P \Rightarrow Q) \Leftrightarrow (\mathcal{D}(P) \wedge \mathcal{D}(Q)) \vee (\mathcal{D}(P) \wedge \neg P) \vee (\mathcal{D}(Q) \wedge Q) \tag{12}$$
$$\mathcal{D}(P \Leftrightarrow Q) \Leftrightarrow \mathcal{D}(P) \wedge \mathcal{D}(Q) \tag{13}$$
$$\mathcal{D}(\exists x \cdot P) \Leftrightarrow (\forall x \cdot \mathcal{D}(P)) \vee (\exists x \cdot \mathcal{D}(P) \wedge P) \tag{14}$$

The statements above can be proved using *FoPCe* (considering the discussion in §2.3 and the precautions stated in the beginning of this section) after unfolding the definitions of the derived logical operators and $\mathcal{D}$.

6 Well-Definedness and Proof

This section contains the main contribution of this paper. The theme of §5 was the well-definedness of individual formulæ. In this section we show how the notion of well-definedness can be integrated into proofs (i.e. sequents and proof rules). In §6.1 we define $\mathcal{D}$ for sequents. We then formally define the notions of a well-defined sequent (§6.2) and a WD preserving proof rule (§6.3) as motivated in §4. In §6.4 we derive the proof calculus $FoPCe_{\mathcal{D}}$, the WD preserving version of *FoPCe*, that we use to preserve well-definedness in a proof. We summarise the results of this section in §6.5.

6.1 Defining $\mathcal{D}$ for Sequents

We now extend our definition of $\mathcal{D}$ to sequents. Observing that the sequent ' $\mathsf{H} \vdash G$ ' is valid iff ' $\vdash \forall \vec{x}.\ \bigwedge \mathsf{H} \ \Rightarrow\ G$ ' is also valid (where '$\forall \vec{x}$' denotes the universal quantification of all free variables occurring in H and G, and '$\bigwedge \mathsf{H}$' denotes the conjunction of all predicates present in H), we extend our well-definedness operator to sequents as follows:

$$\mathcal{D}(\ \mathsf{H} \vdash G\ ) \ \widehat{=}\ \mathcal{D}(\ \forall \vec{x}.\ \bigwedge \mathsf{H} \ \Rightarrow\ G\ ) \tag{15}$$

Note that if we blindly use the definitions (15), (8), (12), and (7) to evaluate '$\mathcal{D}(\ \mathsf{H} \vdash G\ )$' we get a disjunctive predicate that grows *exponentially* with respect to the number of free variables and hypotheses in the sequent. We present ways to overcome this problem in §6.2 and §7.

6.2 Well-Defined Sequents

In §4 we said that the initial sequent of the Validity proof obligation could be considered well-defined since we also prove WD. More generally, we say that a sequent '$\mathsf{H} \vdash G$' is well-defined if we can additionally assume '$\mathcal{D}(\ \mathsf{H} \vdash G\ )$' to be present in its hypotheses. We thereby *encode* the well-definedness of a sequent within its hypotheses. We introduce additional syntactic sugar '$\vdash_{\mathcal{D}}$' to denote such a well-defined sequent:

$$\mathsf{H} \vdash_{\mathcal{D}} G \;\widehat{=}\; \mathcal{D}(\,\mathsf{H} \vdash G\,), \mathsf{H} \;\vdash\; G \tag{16}$$

Re-stating WD and Validity. We may re-state our original proof obligations from §4 in terms of '$\vdash_{\mathcal{D}}$' as follows:

$\mathsf{WD}_{\mathcal{D}}$: $\vdash_{\mathcal{D}} \mathcal{D}(\,\mathsf{H} \vdash G\,)$	$\mathsf{Validity}_{\mathcal{D}}$: $\mathsf{H} \vdash_{\mathcal{D}} G$

Justification. The $\mathsf{WD}_{\mathcal{D}}$ proof obligation is equivalent to the original WD proof obligation since we know from (9) that '$\mathcal{D}(\mathcal{D}(\mathsf{H} \vdash G)) \Leftrightarrow \top$'. To get $\mathsf{Validity}_{\mathcal{D}}$, we add the extra hypothesis '$\mathcal{D}(\,\mathsf{H} \vdash G\,)$' to Validity using the *cut* rule whose first antecedent can be discharged using the proof of WD.

We return to the issue of proving $\mathsf{WD}_{\mathcal{D}}$ and $\mathsf{Validity}_{\mathcal{D}}$ in §7. The rest of this section is concerned with proving well-defined '$\vdash_{\mathcal{D}}$' sequents in general.

Simplifying $\vdash_{\mathcal{D}}$. Directly unfolding (16) introduces the predicate '$\mathcal{D}(\,\mathsf{H} \vdash G\,)$' that, as we have seen in §6.1, grows exponentially when further unfolded. We avoid unfolding '$\mathcal{D}(\,\mathsf{H} \vdash G\,)$' by using the following derived rule instead of (16) to introduce or eliminate $\vdash_{\mathcal{D}}$ from a proof :

$$\frac{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(G), \mathsf{H} \vdash G}{\mathsf{H} \vdash_{\mathcal{D}} G} \vdash_{\mathcal{D}} eqv$$

The $\widehat{\mathcal{D}}$ operator is $\mathcal{D}$ extended for finite sets of formulæ (i.e. $\widehat{\mathcal{D}}(\mathsf{F}) \;\widehat{=}\; \bigcup_{F \in \mathsf{F}} \{\mathcal{D}(F)\}$). Note that $\widehat{\mathcal{D}}(\mathsf{H})$ denotes a set of predicates. The double inference line means that this rule can be used in both directions. The rule $\vdash_{\mathcal{D}} eqv$ says that when proving the validity of a well defined sequent, we may assume that all its hypotheses and its goal are *individually* well-defined.

Proof of $\vdash_{\mathcal{D}} eqv$. We will use the following three derived rules as lemmas in order to prove $\vdash_{\mathcal{D}} eqv$:

$$\frac{\mathcal{D}(P) \vdash P}{\mathcal{D}(\forall x \cdot P) \vdash \forall x \cdot P} \forall_{\mathcal{D}} \qquad \frac{\mathcal{D}(P), \mathcal{D}(Q), P \vdash Q}{\mathcal{D}(P \Rightarrow Q) \vdash P \Rightarrow Q} \Rightarrow_{\mathcal{D}}$$

$$\frac{\mathsf{H}, \mathcal{D}(P), \mathcal{D}(Q), P, Q \vdash R}{\mathsf{H}, \mathcal{D}(P \wedge Q), P \wedge Q \vdash R} \wedge_{\mathcal{D}}$$

The proofs of $\Rightarrow_{\mathcal{D}}$ and $\wedge_{\mathcal{D}}$ in both directions are straightforward using the definitions of $\mathcal{D}$ and the rules of *FoPCe*, and are similar to the proof of $\vdash_{\mathcal{D}} eqv_{simple}$ shown later in this section. The proof of $\forall_{\mathcal{D}}$ though is tricky, but almost identical to the proof of another derived rule $\forall goal_{\mathcal{D}}$ presented in §6.4.

The proof of $\vdash_{\mathcal{D}} eqv$ proceeds as follows. The logical content of the sequent (i.e. the hypotheses and the goal) is first *packed* into the goal of the sequent using the rules of *FoPCe*. This goal is then *unraveled* in parallel with its well-definedness predicate using the three derived rules stated above. Here is the proof:

$$\boxed{\dfrac{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(G), \mathsf{H} \vdash G}{\mathsf{H} \vdash_{\mathcal{D}} G} \vdash_{\mathcal{D}} eqv}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(G), \mathsf{H} \vdash G}{\mathcal{D}(\bigwedge \mathsf{H}), \mathcal{D}(G), \bigwedge \mathsf{H} \vdash G}\,\wedge_{\mathcal{D}}^{|\mathsf{H}|}}{\mathcal{D}(\bigwedge \mathsf{H} \Rightarrow G) \vdash \bigwedge \mathsf{H} \Rightarrow G}\,\Rightarrow_{\mathcal{D}}}{\mathcal{D}(\forall \vec{x}.\ \bigwedge \mathsf{H} \Rightarrow G) \vdash \forall \vec{x}.\ \bigwedge \mathsf{H} \Rightarrow G}\,\forall_{\mathcal{D}}^{|\vec{x}|}}{\mathcal{D}(\mathsf{H} \vdash G) \vdash \forall \vec{x}.\ \bigwedge \mathsf{H} \Rightarrow G}\,(15)}{\mathcal{D}(\mathsf{H} \vdash G), \mathsf{H} \vdash G}\,FoPCe}{\mathsf{H} \vdash_{\mathcal{D}} G}\,(16)$$

In order to save space and the reader's patience, only the important steps of proofs are shown in this paper. Each step is justified using standard proof rules or previously appearing definitions and equivalences. In the proof above, superscripts above the rules $\forall_{\mathcal{D}}$ and $\wedge_{\mathcal{D}}$ indicate the number of applications of these rules. For instance $\forall_{\mathcal{D}}^{|\vec{x}|}$ indicates $|\vec{x}|$ (which is the number of free variables in the original sequent) applications of the rule $\forall_{\mathcal{D}}$. Note that this is allowed since the number of free variables ($|\vec{x}|$) and hypotheses ($|\mathsf{H}|$) contained in a sequent are finite.

In what follows we try to give the reader intuition on why $\vdash_{\mathcal{D}} eqv$ is valid since this cannot be easily seen from the proof just presented. The fact that $\vdash_{\mathcal{D}} eqv$ holds in the downward direction (i.e. if the hypotheses and goal of a sequent are well-defined, then the sequent as a whole is well-defined) is easy to see since hypotheses are weakened. For the upward direction, we present the proof of the simpler case where the well-defined sequent has no free variables and only one hypothesis. The derived rule corresponding to this simple case appears boxed on the right, followed by its proof:

$$\boxed{\dfrac{\mathcal{D}(H), \mathcal{D}(G), H \vdash G}{H \vdash_{\mathcal{D}} G} \vdash_{\mathcal{D}} eqv_{simple}}$$

$$\dfrac{\dfrac{\dfrac{\mathcal{D}(H), \mathcal{D}(G), H \vdash G \quad \dfrac{}{\mathcal{D}(H), \neg H, H \vdash G}\,\neg hyp; hyp \quad \dfrac{}{\mathcal{D}(G), G, H \vdash G}\,hyp}{(\mathcal{D}(H) \wedge \mathcal{D}(G)) \vee (\mathcal{D}(H) \wedge \neg H) \vee (\mathcal{D}(G) \wedge G), H \vdash G}\,FoPCe}{\mathcal{D}(H \Rightarrow G), H \vdash G}\,(12)}{H \vdash_{\mathcal{D}} G}\,(16); (15)$$

From the proof above we can see that, apart from the case where the hypothesis and the goal are individually well-defined, all other possible cases in which the sequent could be well-defined (i.e. the remaining disjuncts of '$\mathcal{D}(H \Rightarrow G)$') can be discharged using the rules in *FoPCe*.

6.3 WD Preserving Proof Rules

We say that a proof rule preserves well-definedness iff its consequent and antecedents only contain well-defined sequents (i.e. $\vdash_{\mathcal{D}}$ sequents). Examples of WD preserving proof rules can be found in §6.4.

We may derive such rules by first using $\vdash_{\mathcal{D}} eqv$ to rewrite $\vdash_{\mathcal{D}}$ sequents in terms of '$\vdash$' and then use *FoPCe* and the properties of $\mathcal{D}$ to complete the proof. Such proofs are discussed in detail in §6.4.

6.4 Deriving $FoPCe_{\mathcal{D}}$

We now have enough formal machinery in place to derive the WD preserving proof calculus $FoPCe_{\mathcal{D}}$. For each proof rule 'r' in *FoPCe* we derive its WD preserving version '$r_{\mathcal{D}}$' that only contains sequents using '$\vdash_{\mathcal{D}}$' instead of '$\vdash$'. Here are the resulting proof rules for $basicFoPCe_{\mathcal{D}}$ that are (apart from the $\vdash_{\mathcal{D}}$ turnstile) identical to their counterparts in *basicFoPCe*:

$$\frac{}{\mathsf{H}, P \vdash_{\mathcal{D}} P}\ hyp_{\mathcal{D}} \qquad \frac{\mathsf{H} \vdash_{\mathcal{D}} Q}{\mathsf{H}, P \vdash_{\mathcal{D}} Q}\ mon_{\mathcal{D}} \qquad \frac{\mathsf{H}, \neg P \vdash_{\mathcal{D}} \bot}{\mathsf{H} \vdash_{\mathcal{D}} P}\ contr_{\mathcal{D}}$$

$$\frac{}{\mathsf{H}, \bot \vdash_{\mathcal{D}} P}\ \bot hyp_{\mathcal{D}} \qquad \frac{\mathsf{H}, P \vdash_{\mathcal{D}} \bot}{\mathsf{H} \vdash_{\mathcal{D}} \neg P}\ \neg goal_{\mathcal{D}} \qquad \frac{\mathsf{H} \vdash_{\mathcal{D}} P}{\mathsf{H}, \neg P \vdash_{\mathcal{D}} Q}\ \neg hyp_{\mathcal{D}}$$

$$\frac{\mathsf{H} \vdash_{\mathcal{D}} P \qquad \mathsf{H} \vdash_{\mathcal{D}} Q}{\mathsf{H} \vdash_{\mathcal{D}} P \wedge Q}\ \wedge goal_{\mathcal{D}} \qquad \frac{\mathsf{H}, P, Q \vdash_{\mathcal{D}} R}{\mathsf{H}, P \wedge Q \vdash_{\mathcal{D}} R}\ \wedge hyp_{\mathcal{D}}$$

$$\frac{\mathsf{H} \vdash_{\mathcal{D}} P}{\mathsf{H} \vdash_{\mathcal{D}} \forall x \cdot P}\ \forall goal_{\mathcal{D}}\ (x\ \underline{\mathsf{nfin}}\ \mathsf{H})$$

$$\frac{}{\mathsf{H} \vdash_{\mathcal{D}} E = E}\ = goal_{\mathcal{D}} \qquad \frac{\mathsf{H} \vdash_{\mathcal{D}} [x := E]P}{\mathsf{H}, E = F \vdash_{\mathcal{D}} [x := F]P}\ = hyp_{\mathcal{D}}$$

The remaining rules, *cut* and $\forall hyp$, need to be reformulated by adding new antecedents (that appear boxed below) to make them WD preserving:

$$\frac{\boxed{\mathsf{H} \vdash_{\mathcal{D}} \mathcal{D}(\mathrm{P})} \qquad \mathsf{H} \vdash_{\mathcal{D}} P \qquad \mathsf{H}, P \vdash_{\mathcal{D}} Q}{\mathsf{H} \vdash_{\mathcal{D}} Q}\ cut_{\mathcal{D}}$$

$$\frac{\boxed{\mathsf{H} \vdash_{\mathcal{D}} \mathcal{D}(\mathrm{E})} \qquad \mathsf{H}, [x := E]P \vdash_{\mathcal{D}} Q}{\mathsf{H}, \forall x \cdot P \vdash_{\mathcal{D}} Q}\ \forall hyp_{\mathcal{D}}$$

These new antecedents are WD sub-goals that need to be discharged when these rules are used in a proof.

The original rules (*cut* and $\forall hyp$) are not WD preserving *since they introduce new predicates and expressions that may be ill-defined* into a proof. Note that the converse is not true. A valid proof rule in *FoPCe* that does not introduce any new formulæ into a proof can be non WD preserving. The following derived proof rule illustrates this:

$$\frac{\mathsf{H} \vdash P \qquad \mathsf{H}, P \vdash Q}{\mathsf{H} \vdash P \wedge Q}$$

Although this rule is valid (it can be proved using $\wedge goal$ and cut) and does not introduce any new formulæ, it does not preserve well-definedness. The first antecedent would be ill-defined in the case where P (say '$pred(x) = 0$') is only well-defined in conjunction with Q (say '$x \neq 0$').

The proofs of the rules of $basicFoPCe_{\mathcal{D}}$ are discussed later in §6.4.

Derived Logical Operators. Once we have derived the rules of $basicFoPCe_{\mathcal{D}}$ stated above, we may use them directly (i.e. without the detour through $\vdash$ sequents) to derive the corresponding WD preserving proof rules for the derived logical operators $\top$, $\vee$, $\Rightarrow$, $\Leftrightarrow$ and $\exists$. The statements of these rules can be found in §4.5.3 of [17]. The only rule here that needs modification is the existential dual of $\forall hyp$ (i.e. $\exists goal$). The resulting proof rules constitute our complete WD preserving proof calculus $FoPCe_{\mathcal{D}}$.

Conditional Definitions. The payoff achieved by using $FoPCe_{\mathcal{D}}$ instead of $FoPCe$ in proofs is that conditional definitions in $FoPCe$ become 'unconditional' in $FoPCe_{\mathcal{D}}$. The '$\vdash_{\mathcal{D}}$' version of the f_{def} rule from §3 is:

$$\frac{}{\vdash_{\mathcal{D}} y = f(\vec{x}) \;\Leftrightarrow\; D^f_{\vec{x},y}}\; f_{def}{}_{\mathcal{D}}$$

The above rule can be derived from f_{def} since $\mathcal{D}\left(\, y = f(\vec{x}) \Leftrightarrow D^f_{\vec{x},y}\right) \Rightarrow C^f_{\vec{x}}$. The above rule differs from f_{def} in that it does not explicitly require the WD condition '$C^f_{\vec{x}}$' in order to be applied. This makes proofs involving partial functions shorter and less tedious to perform. As a result, derived rules such as '$x \neq 0 \vdash pred(x + y) = pred(x) + y$' can also be applied without explicitly having to prove its premise '$x \neq 0$'.

Proofs of $basicFoPCe_{\mathcal{D}}$. The proofs of each rule in $basicFoPCe_{\mathcal{D}}$ that appear in §6.4 follow essentially from $\vdash_{\mathcal{D}} eqv$, the properties of $\mathcal{D}$, and the rules in $FoPCe$. Since some of the proofs are not straightforward we outline some of their major steps in this section as an aid the reader who wants to reproduce them. This section may otherwise be skipped.

The proofs of $hyp_{\mathcal{D}}$, $\bot hyp_{\mathcal{D}}$ and $= goal_{\mathcal{D}}$ are trivial since these rules contain no antecedents. In general, any valid rule having no antecedents is trivially WD preserving.

The proofs for $mon_{\mathcal{D}}$, $contr_{\mathcal{D}}$, $\neg goal_{\mathcal{D}}$, $\neg hyp_{\mathcal{D}}$, and $\wedge hyp_{\mathcal{D}}$ are straightforward and similar in style to the proof of $cut_{\mathcal{D}}$ shown below:

$$\boxed{\frac{\mathsf{H} \vdash_{\mathcal{D}} \mathcal{D}(P) \qquad \mathsf{H} \vdash_{\mathcal{D}} P \qquad \mathsf{H}, P \vdash_{\mathcal{D}} Q}{\mathsf{H} \vdash_{\mathcal{D}} Q}\; cut_{\mathcal{D}}}$$

$$\frac{\dfrac{\mathsf{H} \vdash_{\mathcal{D}} \mathcal{D}(P)}{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(Q), \mathsf{H} \vdash \mathcal{D}(P)}\; (\mathbf{9}); \vdash_{\mathcal{D}} eqv \qquad \dfrac{\mathsf{H} \vdash_{\mathcal{D}} P \qquad \mathsf{H}, P \vdash_{\mathcal{D}} Q}{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(Q), \mathsf{H}, \mathcal{D}(P) \vdash Q}\; cut_{(P)}; \vdash_{\mathcal{D}} eqv}{\dfrac{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(Q), \mathsf{H} \vdash Q}{\mathsf{H} \vdash_{\mathcal{D}} Q}\; \vdash_{\mathcal{D}} eqv}\; cut_{(\mathcal{D}(P))}$$

The proofs of $\forall hyp_{\mathcal{D}}$ and $= goal_{\mathcal{D}}$ require the following additional properties about how $\mathcal{D}$ interacts with the substitution operator:

$$\mathcal{D}([x := E]F) \Rightarrow [x := E]\mathcal{D}(F) \tag{17}$$

$$([x := E]\mathcal{D}(F)) \wedge \mathcal{D}(E) \Rightarrow \mathcal{D}([x := E]F) \tag{18}$$

Both these properties can be proved by induction on the structure of basic formulæ.

The proofs of $\wedge goal_{\mathcal{D}}$ and $\forall goal_{\mathcal{D}}$ are tricky and require rewriting definitions (7) and (8) as follows:

$$\mathcal{D}(P \wedge Q) \Leftrightarrow (\mathcal{D}(P) \wedge (P \Rightarrow \mathcal{D}(Q))) \ \vee \ (\mathcal{D}(Q) \wedge (Q \Rightarrow \mathcal{D}(P))) \tag{19}$$

$$\mathcal{D}(\forall x \cdot P) \Leftrightarrow \exists x \cdot \ (\mathcal{D}(P) \wedge (P \Rightarrow \forall x \cdot \mathcal{D}(P))) \tag{20}$$

Using these equivalences instead of (7) and (8) allows for more natural case splits in these proofs. The proof of $\wedge goal_{\mathcal{D}}$ is similar to the proof of $\forall goal_{\mathcal{D}}$ shown below:

$$\frac{\mathsf{H} \vdash_{\mathcal{D}} P}{\mathsf{H} \vdash_{\mathcal{D}} \forall x \cdot P} \ \forall goal_{\mathcal{D}} \ (x \ \underline{\mathsf{nfin}} \ \mathsf{H})$$

$$\dfrac{\dfrac{\mathsf{H} \vdash_{\mathcal{D}} P}{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(P), \mathsf{H} \vdash P} \vdash_{\mathcal{D}} eqv \quad \dfrac{\dfrac{\dfrac{\mathsf{H} \vdash_{\mathcal{D}} P}{\widehat{\mathcal{D}}(\mathsf{H}), \forall x \cdot \mathcal{D}(P), \mathsf{H} \vdash P} \forall hyp; \vdash_{\mathcal{D}} eqv}{\widehat{\mathcal{D}}(\mathsf{H}), \forall x \cdot \mathcal{D}(P), \mathsf{H} \vdash \forall x \cdot P} \forall goal \ (x \ \underline{\mathsf{nfin}} \ \mathsf{H})}{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(P), \forall x \cdot \mathcal{D}(P), \mathsf{H} \vdash \forall x \cdot P} mon_{(\mathcal{D}(P))}}{\widehat{\mathcal{D}}(\mathsf{H}), \mathcal{D}(P), P \Rightarrow \forall x \cdot \mathcal{D}(P), \mathsf{H} \vdash \forall x \cdot P} \Rightarrow hyp}{\widehat{\mathcal{D}}(\mathsf{H}), \exists x \cdot (\mathcal{D}(P) \wedge (P \Rightarrow \forall x \cdot \mathcal{D}(P))), \mathsf{H} \vdash \forall x \cdot P} \exists hyp; \wedge hyp}{\mathsf{H} \vdash_{\mathcal{D}} \forall x \cdot P} \vdash_{\mathcal{D}} eqv; (20)$$

Note that the rules $\Rightarrow hyp$ and $\exists hyp$ above are the standard left hand sequent calculus rules for implication and existential quantification. We now summarise the results of this section.

6.5 Summary

In this section we have shown how the notion of well-definedness can be integrated into proofs by extending the definition of $\mathcal{D}$ to sequents (§6.1), and characterising well-defined '$\vdash_{\mathcal{D}}$' sequents (§6.2). We have derived the proof rule $\vdash_{\mathcal{D}} eqv$ (§6.2) that allows us to freely move between ordinary '$\vdash$' sequents and well-defined '$\vdash_{\mathcal{D}}$' sequents in proofs. We have formally derived the proof calculus $FoPCe_{\mathcal{D}}$ (§6) whose rules preserve well-definedness. The rules of $FoPCe_{\mathcal{D}}$ are identical to those in $FoPCe$ except for three cases (cut, $\forall hyp$ and its dual $\exists goal$) where additional WD sub-goals need to be proved.

We now return to the practical issue of proving the WD and Validity proof obligations introduced in §4.

7 Proving $\mathsf{WD}_{\mathcal{D}}$ and $\mathsf{Validity}_{\mathcal{D}}$

In §6.2 we re-stated our original proof obligations from §4 in terms of '$\vdash_{\mathcal{D}}$' as follows:

$\mathsf{WD}_{\mathcal{D}}: \quad \vdash_{\mathcal{D}} \mathcal{D}(\ \mathsf{H} \vdash G\ )$	$\mathsf{Validity}_{\mathcal{D}}: \quad \mathsf{H} \vdash_{\mathcal{D}} G$

Proving $\mathsf{WD}_{\mathcal{D}}$. In our practical setting we factor out proving $\mathsf{WD}_{\mathcal{D}}$ for each proof obligation *individually* by proving instead that the (source) models used to generate these proof obligations are well-defined. Details on well-definedness of models can be found in [7]. We are guaranteed that all proof obligations generated from a well-defined model are themselves well-defined and therefore do not need to generate or prove the well-definedness of each proof obligation individually. This considerably reduces the number of proofs that need to be done.

Proving $\mathsf{Validity}_{\mathcal{D}}$. From the '$\vdash_{\mathcal{D}}$' turnstile we immediately see that the initial sequent of $\mathsf{Validity}_{\mathcal{D}}$ is well-defined. We have two choices for how to proceed with this proof. We may either use $FoPCe_{\mathcal{D}}$ to preserve well-definedness, or the standard *FoPCe*.

We prefer using the WD preserving calculus $FoPCe_{\mathcal{D}}$ (with additional WD sub-goals) instead of *FoPCe* for *interactive* proofs for three reasons. Firstly, as seen in §4, the assumption that a sequent is well-defined can be used to greatly ease its proof. Secondly, the extra WD sub-goals require only minimal additional effort to prove in practice, and are in most cases automatically discharged. Thirdly, proving WD sub-goals allows us to *filter out* erroneous ill-defined formulæ entered by the user.

Alternatively, we may use the rule $\vdash_{\mathcal{D}} eqv$ (§6.2) to make the well-definedness assumptions of a '$\vdash_{\mathcal{D}}$' sequent explicit and use the standard proof rules in *FoPCe* to complete a proof. This may not be a prudent way to perform interactive proof, but it allows us to use existing automated theorem provers for *FoPCe* (that do not have any notion of well-definedness) to automatically discharge pending sub-goals, for which we have observed favourable results.

8 Related Work

A lot of work has been done in the area of reasoning in the presence of partial functions. A good review of this work can be found in [13] and [4]. In this section we first describe some approaches that are most relevant to the work presented in this paper and then compare our work to these approaches in §8.1.

The current approaches to reason about the undefined can be classified into two broad categories: those that *explicitly reason* about undefined values using a three-valued logic [16], and those that *avoid* reasoning about the undefined using underspecification [13]. We start with the former.

A well known approach is the Logic of Partial Functions (LPF) [15,6] used by the VDM [14] community. Its semantics is based on three-valued logic [16]. The

resulting proof calculus for LPF can then be used to *simultaneously* prove the validity and well-definedness of a logical statement. A drawback of using LPF (or any other special-purpose proof calculus) is that it differs from the standard predicate calculus since it disallows use of the law of excluded middle (to avoid proving '$3/0 = x \vee 3/0 \neq x$') and additionally requires a second 'weak' notion of equality (to avoid proving '$3/0 = 3/0$'). Additional effort is therefore needed to learn or automate LPF, as for any special-purpose proof calculus, as mentioned in §1.

In PVS [19], partial functions are modelled as total functions whose domain is a predicate subtype. For instance, the partial function '/' is defined as a total function whose second argument belongs to the subtype of non-zero reals. Type-checking then avoids ill-definedness but requires proof. The user needs to prove type correctness conditions (TCCs) before starting or introducing new formulæ into a proof. A shortcoming of this approach is that type-checking requires complicated typing rules [20] and special tool support. This approach additionally blurs the distinction between type-checking (which is usually accepted to be automatically decidable) and proof.

In [7], Behm et al. use a three-valued semantics to develop a proof calculus for B [2]. Its main difference from LPF is that the undefined value, although part of the logical semantics, does not enter into proofs, as explained below. In this approach, all formulæ that appear in a proof need to be proved to be well-defined. Proving well-definedness is similar to proving TCCs in PVS. It has the role of *filtering out* expressions that may be ill-defined. Once this is done, the proof may continue in a *pseudo-two-valued* logic since the undefined value is proved never to occur. The drawback of this approach is similar to that of LPF. Although the proof calculus presented for this pseudo-two-valued logic "is close to the standard sequent calculus" [7], this too is a special-purpose logic. No concrete connection with the standard predicate calculus is evident since this approach, from the start, assumes a three-valued semantics.

In [4], Abrial and Mussat formalise the notion of well-definedness without any detour through a three-valued semantics, remaining entirely within the "syntactic manipulation of proofs" [4] in standard predicate calculus. The resulting well-definedness filter is identical to that in [7]. They formally show how proving statements that passed this filter could be made simpler (i.e. with fewer checks) on the basis of their well-definedness. What is missing in [4] however is a proof calculus (like the one in [7]) that preserves well-definedness, which could additionally be used for interactive proof.

In [8], Berezin et al. also use the approach of filtering out ill-defined statements before attempting to prove them in the automated theorem prover CVC lite. The filter used is identical to the one used in [7] and [4]. Although they too start from a three-valued logic, they show (using semantic arguments) how the proof of a statement that has passed this filter may proceed in standard two-valued logic. Apart from introducing three-valued logic only to reduce it later to two-valued logic, this approach is concerned with purely automated theorem proving and therefore provides no proof calculus that preserves well-definedness to use in

interactive proofs. It is advantageous to preserve well-definedness in interactive proofs (reasons for this are given in §4).

The idea of *avoiding* reasoning about undefined values using underspecification [13] is used in many approaches that stick to using two-valued logic in the presence of partial functions. This is the approach used in Isabelle/HOL [18], HOL [12], JML [9] and Spec# [5]. In this setting, an expression such as '$3/0$' is still a valid expression, but its value is unspecified. Although underspecification allows proofs involving partial functions to be done in two-valued logic, it has two shortcomings. First, (as described in §3) it also allows statements that contain ill-defined terms such as '$3/0 = 3/0$' to be proved valid. In the context of generating or verifying program code, expressions such as '$3/0$' originating from a proved development can lead to a run-time error as described in [9]. Second, doing proofs in this setting may require *repeatedly* proving that a possibly undefined expression (e.g. '$3/x$') is actually well defined (i.e. that '$x \neq 0$') in multiple proof steps.

8.1 Comparison

In this section we compare the approach presented in this paper (§5, §6) with the related work just presented. The work presented here extends the approach of [4] by showing how the notion of well-definedness can be integrated into a proof calculus that is suitable for interactive proof.

The role of proving TCCs in PVS is identical to that of proving well-definedness in our approach (i.e. proving the WD proof obligation and the additional WD sub-goals in $cut_{\mathcal{D}}$, $\forall hyp_{\mathcal{D}}$, and $\exists goal_{\mathcal{D}}$). With regards to its logical foundations, we find the possibility of directly defining *truly* partial functions in our setting more convenient and intuitive as opposed to expressing them as total functions over a restricted subtype. The logical machinery we use is much simpler too since we do not need to introduce predicate subtypes and dependent types for this purpose. Since we use standard (decidable) type-checking we have a clear conceptual separation between type-checking and proof. Although our approach does not eliminate the undecidability of checking well-definedness, it saves type checking from being undecidable.

With respect to B [7] and LPF [6], the approach used here does not start from a three-valued semantics but instead reduces all reasoning about well-definedness to standard predicate calculus. We develop the notion of well-definedness purely on the basis of the syntactic operator '$\mathcal{D}$' and proofs in standard predicate calculus. In §6 we derive a proof calculus preserving well-definedness that is identical to the one presented in [7]. Alternatively, we could have chosen to derive the proof calculus used in LPF in a similar fashion. If our definition of well-defined sequents is modified from the one appearing in §6.2 to

$$\mathsf{H} \vdash_{_{LPF}} G \;\widehat{=}\; \widehat{\mathcal{D}}(\mathsf{H}), \mathsf{H} \vdash G \wedge \mathcal{D}(G) \tag{21}$$

the rules that follow for a proof calculus that preserves '$\vdash_{_{LPF}}$' correspond to those in LPF [6]. The only difference between '$\vdash_{_{LPF}}$' and '$\vdash_{_{\mathcal{D}}}$' is that the latter also assumes the well-definedness of the goal, whereas this has to be additionally proved

for '$\vdash_{LPF}$'. We therefore have a clear basis to compare these two approaches. In LPF, well-definedness and validity of the goal are proved *simultaneously*, whereas in [7] (and also as presented in §4), these two proofs are performed separately, where proving WD acts as a *filter*. Since what is proved is essentially the same, the choice of which approach to use is a methodological preference. We use the latter approach although it requires proving two proof obligations because of four reasons. First, the majority of the WD sub-goals that we encounter in practice (from models and interactive proof steps) are discharged automatically. Second, failure to discharge a proof obligation due to ill-definedness can be detected earlier and more precisely, before effort is spent on proving validity. Third, the structure of '$\vdash_{\mathcal{D}}$' sequents allows us to more directly use the results in [4] and [8] to automate proofs. Fourth, we find $FoPCe_{\mathcal{D}}$ more intuitive to use in interactive proofs since its rules are 'closer' to the standard sequent calculus (only three rules need to be modified with an extra WD sub-goal).

An additional contribution over [7] (and [6]) is that we may, at any time, choose to reduce all our reasoning to standard predicate calculus (using the $\vdash_{\mathcal{D}} eqv$ rule derived in §6.2). This is a choice that could not be taken in [7].

We now compare our work to related approaches that use underspecification [13]. As described in §3, underspecification is the starting point from which we develop our approach. The work presented in this paper can be used to overcome two of the shortcomings the underspecification approach mentioned earlier. First, (as discussed in §4) proving the well-definedness of proof obligations (or of the source model) gives us an additional guarantee that partial (underspecified) functions are not evaluated outside their domain in specifications or program code. This has been recently done along similar lines for Spec# [5] in [21]. Second, (as discussed in §4 and §6.3) preserving well-definedness in proofs allows us to avoid having to prove well-definedness repeatedly, every time we are confronted with a possibly ill-defined expression during proof.

9 Conclusion

In this paper we have shown how standard predicate calculus can be used to reason in a setting with potentially ill-defined expressions by extending it with new syntax and derived rules for this purpose.

The results presented in §6 provide a deeper understanding of reasoning in the context of well-definedness, and its connection with the standard predicate calculus. This work has also resulted in reducing the proof burden in the partial setting by providing better tool support within the RODIN development environment [1] for Event-B since:

- Sequents contain less hypotheses because all well-definedness hypotheses are implicit in well-defined sequents, as described in §6.2.
- Conditional definitions become unconditional as described in §6.4.
- Derived rules contain less preconditions as discussed in §4 and §6.4.
- All proofs can be reduced to proofs in standard two-valued predicate calculus as seen from the $\vdash_{\mathcal{D}} eqv$ rule in §6.2 and discussed in §7.

Acknowledgements

The author would like to thank Jean-Raymond Abrial, Cliff Jones, John Fitzgerald, David Basin, Laurent Voisin, Adam Darvas and Vijay D'silva for their comments and lively discussions.

References

1. Rigorous Open Development Environment for Complex Systems (RODIN) official website, http://www.event-b.org/
2. Abrial, J.-R.: The B-Book: Assigning programs to meanings. Cambridge (1996)
3. Abrial, J.-R.: Modeling in Event B: System and Softtware Design. Cambridge (to appear, 2007)
4. Abrial, J.-R., Mussat, L.: On using conditional definitions in formal theories. In: Bert, D., Bowen, J., Henson, M., Robinson, K. (eds.) B 2002 and ZB 2002. LNCS, vol. 2272, pp. 242–269. Springer, Heidelberg (2002)
5. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, pp. 49–69 (2005)
6. Barringer, H., Cheng, J.H., Jones, C.B.: A logic covering undefinedness in program proofs. Acta Inf. 21, 251–269 (1984)
7. Behm, P., Burdy, L., Meynadier, J.-M.: Well defined B. In: Bert, D. (ed.) B 1998. LNCS, vol. 1393, pp. 29–45. Springer, Heidelberg (1998)
8. Berezin, S., Barrett, C., Shikanian, I., Chechik, M., Gurfinkel, A., Dill, D.L.: A practical approach to partial functions in CVC Lite
9. Chalin, P.: Logical foundations of program assertions: What do practitioners want? In: SEFM, pp. 383–393 (2005)
10. Darvas, Á., Mehta, F., Rudich, A.: Efficient well-definedness checking. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS. Springer, Heidelberg (to appear, 2008)
11. Fitzgerald, J.S., Jones, C.B.: The connection between two ways of reasoning about partial functions. Technical Report CS-TR-1044, School of Computing Science. Newcastle University (August 2007)
12. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, New York (1993)
13. Gries, D., Schneider, F.B.: Avoiding the undefined by underspecification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 366–373. Springer, Heidelberg (1995)
14. Jones, C.B.: Systematic software development using VDM, 2nd edn. Prentice-Hall, Inc., Englewood Cliffs (1990)
15. Jones, C.B.: Reasoning about partial functions in the formal development of programs. Electr. Notes Theor. Comput. Sci. 145, 3–25 (2006)
16. Kleene, S.C.: Introduction to metamathematics. Bibl. Matematica. North-Holland, Amsterdam (1952)
17. Mehta, F.D.: Proofs for the Working Engineer. PhD thesis, ETH Zurich (2008)
18. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
19. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system, January 15 (2001)

20. Owre, S., Shankar, N.: The formal semantics of PVS (March 1999), http://www.csl.sri.com/papers/csl-97-2/
21. Rudich, A., Darvas, Á., Müller, P.: Checking well-formedness of pure-method specifications. In: Formal Methods (FM). LNCS. Springer, Heidelberg (2008)
22. Sutcliffe, G., Suttner, C.B.: The TPTP (Thousands of Problems for Theorem Provers) Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21(2), 177–203 (1998)

A Representative Function Approach to Symmetry Exploitation for CSP Refinement Checking

Nick Moffat[1], Michael Goldsmith[2], and Bill Roscoe[3]

[1] QinetiQ, Malvern, UK and Kellogg College, University of Oxford, UK
nick.moffat@kellogg.ox.ac.uk
[2] Formal Systems (Europe) Ltd and Worcester College, University of Oxford, UK
michael@fsel.com
[3] Oxford University Computing Laboratory, Oxford, UK
bill@comlab.ox.ac.uk

Abstract. Effective temporal logic model checking algorithms exist that exploit symmetries arising from parallel composition of multiple identical components. These algorithms often employ a function *rep* from states to representative states under the symmetries exploited. We adapt this idea to the context of refinement checking for the process algebra CSP. In so doing, we must cope with refinement-style specifications. The main challenge, though, is the need for access to sufficient local information about states to enable definition of a useful *rep* function, since compilation of CSP processes to Labelled Transition Systems (LTSs) renders state information a global property instead of a local one. Using a structured form of implementation transition system, we obtain an efficient symmetry exploiting CSP refinement checking algorithm, generalise it in two directions, and demonstrate all three variants on simple examples.

1 Introduction

Model checking suffers from the state explosion problem, which is the tendency for state space to grow exponentially in size (number of states) as the size of the model (system description in the modelling language) grows. A simple example is the exponential state space growth that can occur when adding parallel components.

A popular approach to combating the state explosion problem is to exploit state space symmetries. This approach has received much attention in the context of temporal logic state-based model checking ([1] contains a survey), but little has been published in the context of refinement checking ("refinement-style model checking") for process algebras.

For temporal logic model checking, effective algorithms exist that exploit symmetries arising from parallel composition of multiple identical components. The most common approach uses a function *rep* from states to representative states and requires full symmetry of the model and the property. We adapt this idea for Communicating Sequential Processes (CSP) [2,3] refinement checking. The main

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 258–277, 2008.

challenge, which may be considered a significant obstacle, is the need for local information about states to enable use of a *rep* function; compiling CSP processes to Labelled Transition Systems (LTSs) makes state information a global property, not a local one.

By exploiting a richer notation than LTSs, namely 'structured machines' (already used internally by the FDR [4] refinement checker for other reasons), we can define a suitable *rep* function. We obtain a refinement checking algorithm that explores a reduced state space efficiently for fully symmetric systems that have parallel components.

We generalise this algorithm in two directions. First we drop the requirement for full symmetry. Second we allow a larger class of property specifications: in the temporal logic model checking context, restricting to symmetric temporal logic property formulae effectively requires that the future behaviour allowed by the formula is always symmetric, regardless of what has happened in the past; in contrast, our second generalisation only needs the specification process (corresponding to a property formula) to express symmetric behaviour starting at the initial state. We restrict attention to refinement in CSP's traces model, which allows one to check safety properties; the algorithms extend to other semantic models.

An earlier paper [5] outlined some of our work aimed at efficient identification of CSP process symmetries, and included an approach to exploit symmetries when refinement checking. The exploitation approach in this paper is different.

Section 2 provides background regarding the process algebra CSP and refinement checking between CSP processes. Section 3 defines CSP process symmetry. Section 4 outlines the representative function approach to symmetry exploitation for temporal logic model checking. Section 5 describes structured machines and briefly describes some syntactic rules for identifying symmetries. Section 6 gives our basic symmetry exploiting refinement checking algorithm and Section 7 extends it in the two directions mentioned above. Section 8 presents experimental results and Section 9 concludes. An appendix contains correctness proofs.

2 CSP Language, Refinement, LTSs and Refinement Checking

2.1 CSP and Refinement

Process algebras such as CSP [2,3] allow systems to be modelled as processes, which may be atomic (for example CSP's STOP process) or may be defined as compositions of other, child, processes using available process operators.

CSP has a variety of process operators, including: interleaving ($|||$); generalised parallel ($\|_X$) and alphabetised parallel ($_X\|_Y$), where processes must synchronise on alphabet X or alphabet $X \cap Y$; internal choice ($\sqcap$) and external choice($\Box$); hiding ($\backslash X$); and renaming($[\![R]\!]$), for relation R on events.

Refinement of a process *Spec* by a process *Impl* amounts to all behaviours (of some kind, such as the finite traces) of *Impl* being behaviours of *Spec*. In the traces semantic model, $\mathcal{T}$, a behaviour is a finite trace the process can perform.

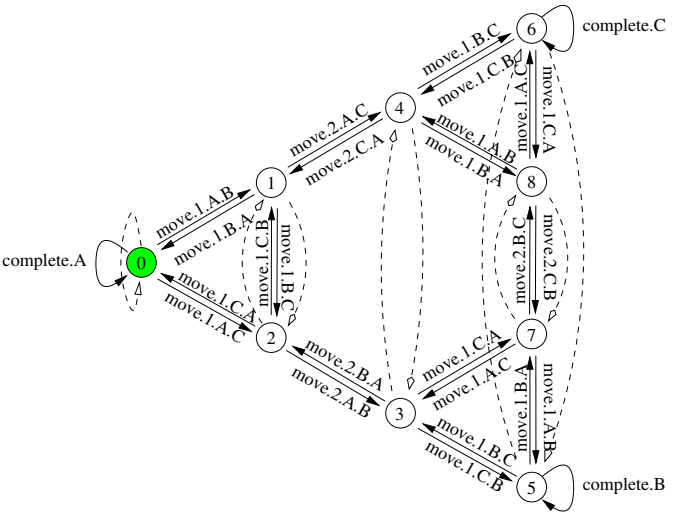


Fig. 1. An LTS for Towers of Hanoi with 3 poles and 2 discs. Dashed arrows show a (`B C`)-bisimulation relation, which can be ignored until Section 3.

2.2 Labelled Transition Systems

A widely used operational form for CSP processes is the Labelled Transition System (LTS). An LTS is a tuple (S, T, s_0) where S is a set of states (sometimes called nodes), $T : S \times \Sigma \times S$ (for universal event set Σ) is a labelled transition relation, and state s_0 is the initial state. An LTS path $\langle s_0, e_1, s_1, \ldots, e_n, s_n \rangle$ has the trace $\langle e_1, \ldots, e_n \rangle$.

The LTS in Figure 1 represents the Towers of Hanoi puzzle with 3 poles and 2 discs. The initial state is shaded. Solid arrows depict the transition relation. An event `move.d.x.y` represents movement of disc `d` from pole `x` to pole `y`, and an event `complete.x` represents an announcement that all discs are on pole `x`. Dashed arrows show a (`B C`)-bisimulation relation, explained in Section 3.

2.3 Refinement Checking

CSP refinement checking algorithms operate over transition systems T_{Spec} and T_{Impl} of a specification process, *Spec*, and an implementation process, *Impl*. Each transition system is a compiled form of the process and supports calculation of the initial state and the set of transitions. Transition system T_{Spec} is required to be an LTS in normal form [3,6], which ensures that no two paths of T_{Spec} with the same trace end at different states.

The usual refinement checking algorithm [6] explores the product space of (*Spec* state, *Impl* state) pairs such that a common trace can take the specification and implementation to the respective states. Exploration starts at the initial state pair and continues until a counterexample has been reached or all successors of reached pairs have been found.

Although it is usual to refer to these as 'pairs' (and we do so throughout), refinement checking algorithms generally record tuples of at least four values; they explore a product space and record extra information as they go, as explained below.

At each pair reached: (i) 'compatibility' of the implementation state with the specification state is checked; (ii) all successor state pairs are added to the set of pairs seen so far. The compatibility test depends on the semantic model used for the check; for the traces model it simply checks that all events labelling outgoing transitions of the implementation state are among the labels on outgoing transitions from the specificiation state.[1]

If an incompatible state pair is reached then a counterexample trace to this pair is recovered by stepping through the implementation transition system backwards until its initial state is reached; this is possible since the identifier of a parent pair is recorded with each newly reached state pair, plus an event from the parent to this pair.

3 CSP Symmetry and Permutation Bisimulations

For process algebras, symmetry acts principally on events/actions; states (equivalently, processes) correspond to particular sets of possible future behaviours. Event permutations lift naturally to state (or process) permutations: throughout the paper, the permutation of a process P by an event permutation σ, written $P\sigma$, is the functional renaming[2] of P according to σ. So $P\sigma$ is the process that can perform event $x\sigma$ whenever P can perform an event x. Also, $=_{\mathcal{T}}$ denotes traces equivalence.

3.1 Algebraic and Denotational Permutation Symmetry

Perhaps the simplest definition of CSP symmetry is in the algebraic semantics. Let σ be any permutation of events in some universal event set Σ, where we insist that $\tau\sigma = \tau$ (i.e., that the special CSP event τ, denoting an internal action, is unaffected by σ). Then we say that a process P is σ-*symmetric* in the traces semantic model, $\mathcal{T}$, when $P =_{\mathcal{T}} P\sigma$.

Notice that we do not restrict σ to preserve channels: we allow permutations that map, say, $a.2$ to $b.44$. However, we may anticipate that a common form of event permutation will be the canonical lifting of a datatype permutation: for example, if δ is a permutation of a datatype D and c is a CSP channel carrying data of type D, then the canonical lifting of δ to an event permutation σ maps events $c.x$ to $c.(x\sigma)$. When events have complex datatypes, the canonical lifting

[1] Internal transitions, labelled by special event τ, are removed from the specification transition system by normalising it, which ensures that no two specification states are reachable by the same trace. Pair (u, v') is treated as a successor to (u, v) if v has a τ transition to v'. For details, see [3] or [6].

[2] Injective functional renaming is defined on page 87 of [3]. Equivalently, we may write $P\sigma$ as $P[\![\sigma]\!]$ using CSP's relational renaming operator.

applies the datatype permutation to all fields having that type. For succintness, we will sometimes represent an event permutation σ by a datatype permutation, in which case σ is understood to be the canonical lifting of this datatype permutation.

The equivalent denotational definition of CSP symmetry is also straightforward: process P is σ-symmetric in $\mathcal{T}$ if the set value that P denotes in $\mathcal{T}$ – the set of finite traces of P – is itself symmetric according to σ, that is, if the set of permuted elements of this set (where each element is permuted by σ lifted to traces) is the set itself.

3.2 Operational Permutation Symmetry

Before defining LTS symmetries we remark that, as one would expect, permutation symmetries of LTSs imply the same symmetries of the processes they represent (though structurally asymmetric LTSs can represent symmetric processes).

Our definition of LTS symmetries uses the more general notion of *permutation bisimulations*, or *pbisims* for short, which were introduced in [5]. Permutation bisimulation extends the classical notion of (strong) bisimulation [7,8]. For event permutation σ, a binary relation R over the nodes S of an LTS L is a σ-*bisimulation* if R is a σ-simulation of L and R^{-1} is a σ^{-1}-simulation of L. Permutation simulation extends the classical notion of simulation: classical simulation requires that (1) if $pRp' \wedge p \xrightarrow{a} q \in L$, then $\exists\, p' \xrightarrow{a} q' \in L$ s.t. qRq'; and (2) $\forall\, p \in S, \exists\, p' \in S$ s.t. pRp'; instead, σ-simulation requires $p' \xrightarrow{a\sigma} q'$ in the consequent of the first condition.

We treat τ events the same way as visible events; when $a = \tau$ we require that $p' \xrightarrow{\tau} q'$ (recall that our event permutations do not affect τ). A possible generalisation is to consider the permutation analogue of weak bisimulation [8]. Our use of strong bisimulation admits fewer symmetries.

Two nodes are σ-*bisimilar* if there is a σ-bisimulation that relates them. Permutation bisimilarity captures the equivalence of processes represented by LTS nodes in the following sense: if node x is σ-*bisimilar* to node y, then the process represented by y equals (in $\mathcal{T}$) the process represented by x (P, say) renamed by σ (i.e., $P\sigma$).

LTS symmetry can now be defined in terms of permutation bisimulation: for permutation σ, an LTS L for a process P is σ-*symmetric* iff some σ-bisimulation relates L's initial state s_0 to itself (i.e., s_0 is σ-bisimilar to itself). The LTS of Figure 1 is (`B C`)-symmetric as the (`B C`)-bisimulation shown relates the initial node to itself.

3.3 Group Symmetry

The above definitions lift easily to group symmetry, as follows. Let G be a group of event permutations. Then a process P, or LTS L, is G-*symmetric* if it is σ-symmetric for each σ in G. (It is clearly sufficient to be σ-symmetric for each of a set of generators of G.)

4 Symmetry and Temporal Logic Model Checking

This section summarises the temporal logic model checking problem and outlines what may be called the "representative function" approach to symmetry exploitation in that context, broadly following the presentation in [1].

A *Kripke structure* over a set AP of atomic propositions is a tuple $M = (S, R, L, S_0)$ where: (1) S is a non-empty finite set of states; (2) $R \subseteq S \times S$ is a total transition relation; (3) $L : S \to 2^{AP}$ is a mapping that labels each state in S with the set of atomic propositions true in that state; and (4) $S_0 \subseteq S$ is a set of initial states. Temporal logic model checking determines whether a given Kripke structure M satisfies a given formula ϕ expressed in some temporal logic (often CTL* or one of its sub-logics LTL or CTL); this is denoted $M \vDash \phi$ and amounts to ϕ holding in each initial state of M.

The representative function approach to symmetry exploitation is applicable with *symmetric* formulae ϕ w.r.t. a group G of *automorphisms* of M (which are state permutations that preserve the transition relation R). A symmetric CTL* formula ϕ w.r.t. a group G of state permutations is one where, for every maximal propositional subformula f in ϕ, f holds in a state s iff it holds in state $\lambda(s)$ for each λ in G. So, symmetric formulae are such that the validity of each maximal propositional subformula is unaffected by permutations in G.

Further, this approach requires that M represents a parallel composition of identical components and that each element of G permutes the values of state variables according to a permutation of component indices.

The idea is to use a 'representative' function, usually called *rep*, chosen according to a symmetry group G where ϕ is known to be symmetric w.r.t. G. This function maps each state s of the Kripke structure to a representative state $rep(s)$ in the same G-orbit as s, where G-orbits are equivalence classes induced by the relation "is related to by some permutation in G". That is, the function *rep* maps each state to a representative state to which it is related by some permutation in G.

A *quotient* Kripke structure $M_G = (S_G, R_G, L_G, S^0_G)$ is generated where: $S_G = \{rep(s) \mid s \in S\}$, $R_G = \{(rep(s), rep(s')) \mid (s, s') \in R\}$, $L_G(rep(s)) = L(rep(s))$, $S^0_G = \{rep(s) \mid s \in S_0\}$. The quotient structure is then checked against the original formula ϕ. It has been proved that $M \vDash \phi$ iff $M_G \vDash \phi$ [9,10]. The quotient check is up to $n!$ times faster than the original, for n identical components, and can consume significantly less memory.

5 Structured Machines and Their Symmetries

5.1 Structured Machines

A *structured machine* represents an LTS as an operator tree with a CSP process operator at each non-leaf node and an LTS at each leaf. Alphabets are associated with child nodes as appropriate for the parent node's CSP operator (i.e., according to the number of operand alphabets). Structured machines reflect an

upper part of a process expression's algebraic structure. They are called *configurations* in [3,6]. They can be much smaller than equivalent LTSs, being linear in the number of component processes of a parallel composition; they can often be operated on very efficiently.

The example in Figure 2 represents a process $P = \|\ p : PEGS \bullet [interface(p)]\ POLE(p)$ for a datatype $PEGS = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and alphabet- and process-valued functions $interface/1$ and $POLE/1$. Their definitions are not shown, but LTSs for the leaf processes $POLE(\mathtt{A})$ etc. are depicted in the right-hand portion of Figure 2. The initial node of each leaf LTS is shaded. The same process P is represented explicitly by the LTS of Figure 1.

For simplicity, we consider only single-configuration processes, which has the effect of allowing only a subset of CSP process operators outside recursive definitions: parallel operators, hiding and renaming. In practice many processes have this form.

A structured machine with a top level parallel operator has states in tuple form – each component denotes the local state of a particular leaf LTS. Hence the initial state of the 3-pole Towers of Hanoi structured machine in Figure 2 is (0,0,0), since each leaf starts in its local state 0. (Alternatively, we could write (1:0,2:0,3:0) but we omit the leaf identifiers.) Subsequent states of this machine are reached by the leaves evolving according to their local states, synchronised with each other on their respective *interface* alphabets. For example, initial state

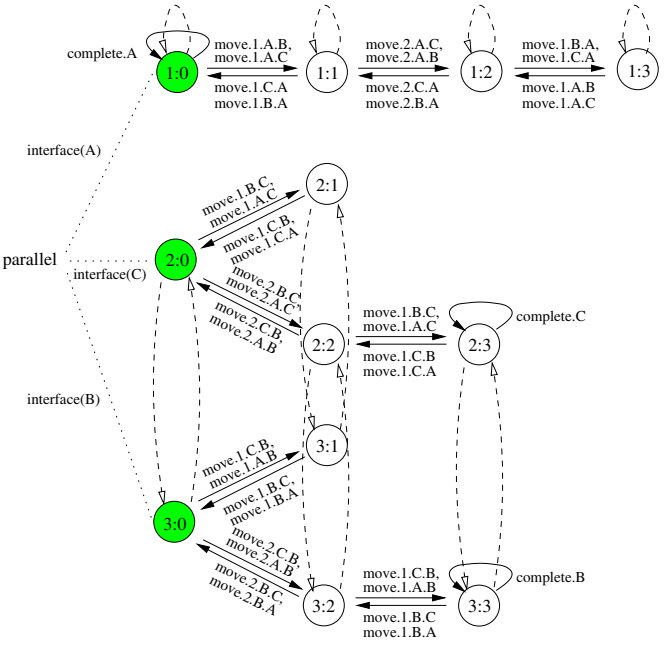


Fig. 2. A structured machine for Towers of Hanoi with 3 poles and 2 discs, with alphabetised parallel at the root. Dashed arrows show a (B C)-bisimulation on LTS nodes.

(0,0,0) has a transition labelled `move.1.A.B` to state (1,0,1); leaf 2 represents pole `C` and is not involved in this transition.

5.2 Structured Machine Symmetries

Symmetries of a structured machine can be represented conveniently using permutation bisimulations between the nodes of its leaf LTSs, as demonstrated by the (`B C`)-bisimulation in Figure 2. A single permutation bisimulation may relate nodes of a single leaf LTS, or nodes of different leaf LTSs. Permutation bisimulations can often be found by exploiting the structure of CSP process expressions, as explained below. Operational and algebraic approaches to checking symmetries and permutation bisimulations were discussed briefly in [5]. The algebraic approach is well suited to efficient identification of structured machine symmetries, so it is described here.

Table 1 expands the table in [5]. It gives a selection of rules that relate trace symmetries of processes to those of sub-processes and alphabets. Due to space limitations, Table 1 is incomplete and we omit our proofs of these results. Throughout, σ is taken to be an event permutation. For an alphabet X (A, H or $A(i)$ in the table), $X\sigma$ denotes the set $\{x\sigma \mid x \in X\}$. In rules 9-13, σ permutes indices in the set I and events according to the corresponding canonical lifting.

Rules 4 and 5 are alternative instances of rule 10 for two sub-processes: rule 4 is obtained when σ maps P to P and Q to Q, and rule 5 is obtained when σ swaps P and Q; rechristening P as $P(1)$ and Q as $P(2)$, the distinction is how σ acts on the indices 1 and 2 in rule 10, i.e. on whether σ maps 1 to 1 and 2 to 2,

Table 1. Some exact (1 and 2) and sufficient (3-13) conditions for CSP process symmetry

	Proc	$(Proc)\sigma =_{\mathcal{T}} Proc$	*Explanation of Proc*
1	$STOP$	$True$	$STOP$ has only empty trace
2	$?x : A \rightarrow P(x)$	$A\sigma = A \wedge \forall x \in A, P(x\sigma) =_{\mathcal{T}} P(x)\sigma$	Accept x in A, become $P(x)$
3	$P \,\Box\, Q$	$P\sigma =_{\mathcal{T}} P \wedge Q\sigma =_{\mathcal{T}} Q$	External choice of P and Q
4	$P \underset{A}{\parallel} Q$	$P\sigma =_{\mathcal{T}} P \wedge Q\sigma =_{\mathcal{T}} Q \wedge A\sigma = A$	P and Q synchronised on A
5	$P \underset{A}{\parallel} Q$	$P\sigma =_{\mathcal{T}} Q \wedge Q\sigma =_{\mathcal{T}} P \wedge A\sigma = A$	P and Q synchronised on A
6	$P \,;\, Q$	$P\sigma =_{\mathcal{T}} P \wedge Q\sigma =_{\mathcal{T}} Q$	P then (on termination) Q
7	$P \setminus H$	$P\sigma =_{\mathcal{T}} P \wedge H\sigma = H$	P with events in H hidden
8	$P[\![R]\!]$	$\exists \rho \bullet P\rho =_{\mathcal{T}} P \wedge$ $\forall a \in \alpha(P), aRb \Rightarrow (a\rho)R(b\sigma^{-1})$	P renamed by event relation R
9	$\|\|\|_{i \in I} P(i)$	$\forall i \in I, P(i\sigma) =_{\mathcal{T}} P(i)\sigma$	Interleaving of all '$P(i)$'s
10	$\underset{A}{\parallel}{}_{i \in I} P(i)$	$\forall i \in I, P(i\sigma) =_{\mathcal{T}} P(i)\sigma \wedge A\sigma = A$	Generalised parallel of '$P(i)$'s
11	$\parallel_{i \in I} (A(i), P(i))$	$\forall i \in I, P(i\sigma) =_{\mathcal{T}} P(i)\sigma \wedge A(i\sigma) = A(i)\sigma$	'$P(i)$'s synchronised on '$A(i)$'s
12	$\Box_{i \in I} P(i)$	$\forall i \in I, P(i\sigma) =_{\mathcal{T}} P(i)\sigma$	External choice of '$P(i)$'s
13	$\sqcap_{i \in I} P(i)$	$\forall i \in I, P(i\sigma) =_{\mathcal{T}} P(i)\sigma$	Internal choice of '$P(i)$'s

or swaps 1 and 2. Rule 3 is an instance of rule 12. In this way, specialised rules can be derived easily from rules 9-13. Rule 8 uses 'exact alphabet' function α.

Rules 9-13 allow one to infer symmetries that are (liftings of) index permutations. These rules can be generalised, replacing $\forall\, i \in I,\ P(i\sigma) =_{\mathcal{T}} P(i)\sigma$ by $\exists$ index permutation $\rho \bullet \forall\, i \in I,\ P(i\rho) =_{\mathcal{T}} P(i)\sigma$, where ρ permutes indices and σ permutes events.

Most of the rules are deliberately approximate. Informally, they only allow 'easy' symmetries to be identified – symmetries one would expect to hold 'at first glance'. This helps to make them simple and easy to implement. Reasoning with such rules will generally miss some symmetries, but we expect they would find most that arise in practice. Some approximation is necessary, as finding all symmetries would in general be too computationally demanding.

One approach to cope with recursive definitions would be to calculate conditions iteratively and terminate on reaching a fixed point. This would require some supporting theory to argue termination and perhaps uniqueness of the fixed point. We take the simpler approach of identifying symmetry of recursive processes operationally [5], by examining transition systems (LTSs, in fact) that represent them.

We have developed a prototype tool which implements extended versions of these rules, for deciding whether any given processes $Proc_1$ and $Proc_2$ are mutually permutation symmetric by a given event permutation σ, that is, whether $Proc_1\sigma =_{\mathcal{T}} Proc_2$. (It is straightforward to extend the rules in this way.) By choosing $Proc_1 = P(x)$ and $Proc_2 = P(x\sigma)$, the extended rules can also be used for checking *permutation transparency* conditions $P(x)\sigma =_{\mathcal{T}} P(x\sigma)$. Such conditions occur at lines 2 and 9-13 of Table 1.

The most significant rules for this paper are those for the replicated parallel operators: rows 9-11 in the table. This is because structured machines with these operators have effective state spaces with states being tuples of local states, one per child machine. Sections 6 and 7 will define *rep* functions on such tuple states.

An alternative, promising approach to finding permutation transparencies (and so symmetries) is to look for data independence (d.i.) [11] of a parametrized process expression $P(x)$ in the type X, say, of its parameter. This is because d.i. – a simple syntactic property – implies transparency with respect to all permutations of the type. It appears possible to liberalise the notion of data independence to yield a syntactic characterisation of a large class of transparent processes: one would remove conditions (notably banning of parallel composition indexed by the d.i. type) designed to prevent d.i. processes 'counting' the datatype. Once one has identified transparency syntactically – using standard d.i. or a liberalised version – one could deduce symmetries using the rules above. This is motivated further in [5], in particular for d.i. index sets.

The algebraic rules in Table 1 can be extended to yield a compiled representation of the process as a structured machine, plus permutation bisimulation relations on the nodes of its leaf LTSs (not merely knowledge of whether the process is symmetric). Such permutation bisimulations will justify the *rep* functions defined in the next sections.

6 Basic Symmetry Exploiting Algorithm

Recall that symmetry of a CTL* formula f w.r.t. group G means that f never discriminates between mutually symmetric behaviours, regardless of the number of steps already taken. The corresponding condition on a specification process is that it is G-symmetric *in each state* (each process to which it can evolve is G-symmetric); if this holds we say the specification process is *universally G-symmetric*. A specification transition system (LTS or structured machine) is universally G-symmetric if each of its states (LTS nodes or tuples of nodes) is G-symmetric, implying universal G-symmetry of the represented process.

The *product space* for a specification *Spec* with states S_{Spec} and an implementation *Impl* with states S_{Impl} is the subspace of $S_{Spec} \times S_{Impl}$ reachable under lock-step synchronisation on all visible events. This state space is explored during a standard refinement check; each 'state' of the product space is really a state pair (u, v), say, where u is a specification state and v is an implementation state. A *path* through a transition system is an alternating sequence $\langle s_0, e_1, s_1, \ldots, e_n, s_n\rangle$ of states and events, starting and ending with states such that for each $0 \leqslant i < n$, there is a transition from s_i to s_{i+1} labelled e_{i+1}.

A *twisted path* through a *Spec*-*Impl* product space is a sequence $\langle s_0, e_1, \sigma_1, s_1, \ldots, e_n, \sigma_n, s_n\rangle$ of (product) states, events and permutations, starting and ending with states, with the following well formedness condition between successive states: $\forall\, 0 \leqslant i < n$, there is a product space transition labelled e_{i+1} from $s_i = (u_i, v_i)$ to $\mathit{pre}\text{-}s_{i+1} \equiv s_{i+1}\sigma_{i+1}^{-1} = (u_{i+1}\sigma_{i+1}^{-1}, v_{i+1}\sigma_{i+1}^{-1})$. Intuitively, non-trivial permutations σ 'twist' the search away from paths the usual refinement checking algorithm would follow.

Given a function *repPair* from state pairs to state pairs, a *repPair*-*twisted path* is a twisted path $\langle s_0, e_1, \sigma_1, s_1, \ldots, e_n, \sigma_n, s_n\rangle$ such that $\forall\, 0 < i \leqslant n$, $s_i = \mathit{repPair}(\mathit{pre}\text{-}s_i)$, where $\mathit{pre}\text{-}s_i \equiv s_i\sigma_i^{-1}$. (We let *repPair* return a permutation too, which this definition ignores.)

6.1 TwistedCheck

The symmetry exploiting algorithms will be defined in terms of a curried function TwistedCheck (see Figure 3) parametrized by a function *repPair*. Ignoring counterexample recovery for the moment, TwistedCheck(*repPair*) differs from the usual refinement checking algorithm (Section 2.3) as follows: during exploration, instead of recording a reached state pair (u, v), record $(\mathit{rep}_u, \mathit{rep}_v)$ where $(\mathit{rep}_u, \mathit{rep}_v, \sigma) = \mathit{repPair}(u, v)$. Note that TwistedCheck(*repPair*) does not need *Spec* or *Impl* G-symmetry.

TwistedCheck(*repPair*) explores the *Spec*-*Impl* product space by following *repPair*-twisted paths – each non-trivial permutation σ returned by *repPair* redirects the search to continue from `(repnext_u,repnext_v)` instead of from `(next_u,next_v)`. Each such σ is recorded for counterexample recovery.

A *bad state pair*, and a *bad event* from that pair, are a pair (u, v) and event e where *Impl* state v has an outward transition labelled e but *Spec* state u does not. We generalise the notion of bad event: a *bad trace* from a bad state pair (u, v) is a trace t such that *Impl* state v can perform t but *Spec* state u cannot.

```
TwistedCheck(repPair)(T_Spec,T_Impl)
Input:  Normal Spec transition system T_Spec with states S_Spec
        Impl structured machine T_Impl with states S_Impl
        repPair: S_Spec x S_Impl -> S_Spec x S_Impl x G
Output: A repPair-twisted counterexample or 'REFINES'

function recover2(state,vparent,e,π)
  if defined(vparent) then
    (u,v,vparent2,e2,σ) := Seen[vparent];
    return recover2(v,vparent2,e2,σπ)^<eπ>;
  else
    return <>;
  endif
end

Seen := {(init(T_Spec),init(T_Impl),undef,undef,1)}; Done := {};

while Seen - Done is not empty do
  Choose some (u,v,vparent,event,π) from Seen-Done;
  if v is compatible with u then
    foreach transition (v,e_v,next_v) in T_Impl do
      e_u := e_v;
      Let next_u be unique such that (u,e_u,next_u) ∈ T_Spec;
      (repnext_u,repnext_v,σ) := repPair(next_u,next_v);
      Put (repnext_u,repnext_v,v,e_v,σ) in Seen if
        no tuple in Seen has same first two values;
    endfor
  else
    bad := an event possible for v but not for u;
    print recover2(v,vparent,event,π)^bad; abort;
  endif
  Done := union(Done,(u,v,vparent,event,π));
endwhile

print 'REFINES';
```

Fig. 3. Twisted refinement checking algorithm for traces refinement. Underlining shows the differences compared with the usual refinement checking algorithm.

A *counterexample trace* is a trace to a bad state pair, extended by a bad trace from that pair. It is easy to see that the counterexample traces are exactly the *Impl* traces that are not *Spec* traces.

Define *recover*($\langle$path$\rangle$) and *recover*2($\langle$*repPair*-twisted path$\rangle$) as follows:

$$\mathit{recover}(\langle s_0, e_1, s_1, \ldots, e_n, s_n\rangle) = \langle e_1, \ldots, e_n\rangle$$
$$\mathit{recover}2(\langle s_0, e_1, \sigma_1, s_1, \ldots, e_n, \sigma_n, s_n\rangle) = \langle e_1\sigma_1\sigma_2 \ldots \sigma_n, \ldots,\ e_n\sigma_n\rangle$$

So *recover*(p) is the trace of events along path p, and *recover*2 also yields a trace. Let a *repPair trace to state pair* s be the result of applying *recover*2 to a *repPair*-twisted path r to s.

A *repPair counterexample trace* is then a *repPair* trace to a bad state pair, extended by a bad trace from that pair. Examination of Figure 3 shows that on reaching a bad pair (u, v) the line 19 condition fails and TwistedCheck(*repPair*) effectively applies *recover*2 to a *repPair*-twisted path to (u, v), extends the result by a bad event, and so obtains a *repPair* counterexample trace.

6.2 SymCheck1

Suppose a function *rep* maps each implementation state v to a representative in the G-equivalence class of v, for some event permutation group G. Define SymCheck1 to be TwistedCheck(*repPair*1) where *repPair*1 is defined in terms of a function *sortRep*:

$$repPair1(u, v) = (u, rep(v), \sigma), \text{ some } \sigma \text{ in } G \text{ s.t. } v\sigma = rep(v)$$
$$rep = sortRep$$

SymCheck1 explores the *Spec-Impl* product space by following *repPair*1-twisted paths. Recall that universal G-symmetry of a specification transition system T_{Spec} means that each state u of T_{Spec} is G-symmetric, i.e. each u is such that $u\sigma = u$ for all σ in G. So, for state u of T_{Spec} and state v of T_{Impl}, *repPair*1 maps state pair (u, v) to $(u, rep(v), \sigma)$ [for some σ in G s.t. $v\sigma = rep(v)$] $= (u\sigma, v\sigma, \sigma)$ [using universal G-symmetry of T_{Spec} and that $v\sigma = rep(v)$] $= (u\sigma_{u,v}, v\sigma_{u,v}, \sigma_{u,v})$ for $\sigma_{u,v} = \sigma$. The significance of this is that Theorem 2, proved in the appendix, applies:

Theorem 2. *Let G be a group of event permutations and suppose Spec and Impl have G-symmetric transition systems T_{Spec} and T_{Impl} respectively. Suppose function repPair maps each state pair (u, v) to $(u\sigma_{u,v}, v\sigma_{u,v}, \sigma_{u,v})$ for some $\sigma_{u,v}$ in G. Then Spec $\sqsubseteq_{\mathcal{T}}$ Impl has a counterexample trace t iff Spec $\sqsubseteq_{\mathcal{T}}$ Impl has a repPair counterexample trace t.*

So, SymCheck1 eventually finds a *repPair* counterexample trace exactly when the refinement does not hold, and this will be a counterexample trace. If the exploration order is breadth-first, the counterexample found will clearly have minimal length.

6.3 Method *sortrep*

It remains to define a suitable function *rep* that maps each implementation state v to some G-equivalent representative. Given a group G, an implementation structured machine T_{Impl} with n leaves and a state $v = (v_1, \ldots, v_n)$ of T_{Impl}, we describe a method of calculating a representative $rep(v)$ and a permutation σ in G such that $v\sigma = rep(v)$. This method and an alternative defined later both rely on knowledge of the permutation bisimulations between the nodes of T_{Impl} and in particular the σ-bisimulations for σ in G. The method *sortRep* is fast but, as discussed below, it needs all pbisims to have a simple form. Furthermore, G must be a full symmetry group.

Suppose process P is a parallel composition, by some parallel operator *op*, of $n > 1$ processes $P(id_1), \ldots, P(id_n)$ represented by LTSs $L(1), \ldots, L(n)$. Then P can be represented by a structured machine S having a simple form if *op* is interleaving, generalised parallel on some shared G-symmetric alphabet A, or alphabetised parallel on G-transparent alphabets (so each process $P(id)$ is synchronised with the others on an alphabet $A(id)$, where $A(id\sigma) = A(id)\sigma$ for each σ in G). In such cases, P is representable by structured machine S having top level operator *op* and children $L(1), \ldots, L(n)$. As stated earlier, Figure 2 gives an example for Towers of Hanoi with 3 poles and 2 discs.

A pbisim p_σ that relates leaf nodes of a structured machine is a *simple swap pbisim for leaf indices* i and j if p_σ relates each $i{:}m$ (shorthand for a node m of LTS i) to $j{:}m$, each $j{:}m$ to $i{:}m$ and, for $k \notin \{i, j\}$, each $k{:}m$ to $k{:}m$. For example, the `(B C)`-bisimulation depicted in Figure 2 is a simple swap pbisim for leaves 2 and 3.

Consider arbitrary state $v = (v_1, \ldots, v_n)$ of such a structured machine S having simple swap σ-bisimulation $p_{i,j}$ for $i < j$. Then applying $p_{i,j}$ to v has the effect of swapping the values at indices i and j of v, and not changing other values, to yield the state $v' = (v_1, \ldots, v_{i-1}, v_j, v_{i+1}, \ldots, v_{j-1}, v_i, v_{j+1}, \ldots, v_n)$. By the generalised form of rule 9, 10, or 11 of Table 1 (depending on the operator *op*), state v is σ-bisimilar to v' since each component of v is σ-bisimilar to a component of v'. In the case of Figure 2, each structured machine state (x, y, z), for some local states x, y and z of the respective leaf LTSs, is related by the depicted pbisim to (x, z, y).

Suppose *PBISIMS* is a set of simple swap pbisims for a structured machine having n leaves, and suppose *SUB* is a subset of $\{1, \ldots, n\}$. Then *PBISIMS* is a *full set of simple swap pbisims for SUB* if for each $i, j \in SUB$ there is a simple swap pbisim $p_\sigma \in PBISIMS$ for i and j, with p_σ a σ-bisimulation relation. Let G be the group generated by such permutations σ. Then any permutation of component states v_i of $v = (v_1, \ldots, v_n)$ with indices in *SUB* yields a G-bisimilar state because each such permutation is the composition of a sequence of transpositions, each of which maps to a G-bisimilar state. The method *sortRep*, for such a set *SUB*, sorts the components of a state $v = (v_1, \ldots, v_n)$ that have indices in *SUB* and leaves the others unchanged; the resulting state is G-bisimilar to v due to the above reasoning.

There is scope for defining variants of this method that apply to more processes. In particular, it would be straightforward to cope with structured machines whose pbisims correspond to multiple simultaneous index swaps – such as (1 2)(5 6). One could still use a fast sort-based method: sort a subset of the local state values (say, v_1 and v_2) and apply a corresponding permutation to the other values (v_5 and v_6 in this example).

7 Extensions

Two extended algorithms are described. SymCheck2 uses a more general *rep* function that applies to a larger class of implementation processes than does

sortRep. SymCheck3 is more general still, requiring only G-symmetry of the *Spec* transition system instead of universal G-symmetry.

7.1 SymCheck2

Define SymCheck2 to be TwistedCheck(*repPair*2), where *repPair*2 uses a more general *rep* function:

$$repPair2(u, v) = (u, rep(v), \sigma), \text{ some } \sigma \text{ in } G \text{ s.t. } v\sigma = rep(v)$$
$$rep = genRep$$

SymCheck2 explores the *Spec*-*Impl* product space by following *repPair*2-twisted paths. Compared to SymCheck1, SymCheck2 uses *genRep* (defined below) in place of *sortRep*. Similarly to SymCheck1, Theorem 2 justifies use of SymCheck2 to find counterexamples when the *Spec* transition system is universally G-symmetric and the *Impl* transition system is G-symmetric. Both SymCheck1 and SymCheck2 require universal G-symmetry of *Spec* to ensure that Theorem 2 applies. The practical difference is that SymCheck2 is less restrictive than SymCheck1 about the form of the *Impl* transition system and its known permutation bisimulations.

Method *genRep*. As already mentioned, this method is more general than *sortRep*. It works with any set of *Impl* permutation bisimulations such that, for each leaf index i, each pbisim p relates all nodes of $LTS(i)$ to nodes of a distinct $LTS(j)$, and each such $LTS(j)$ node is the image of some $LTS(i)$ node by p, where j depends on the pbisim (and could be the same as i). That is, we require each pbisim p to be the union of bijections $\{p_1, \ldots, p_n\}$ with each p_i having domain the nodes of $LTS(i)$ and range the nodes of some distinct $LTS(j)$. We call such pbisims *uniform*. (Uniformity is a natural condition, indeed all pbisims calculated using our extended Table 1 rules are uniform, and composition of pbisims preserves uniformity.)

The method *genRep* calculates each state $(v'_1, \ldots, v'_n)$ to which $v = (v_1, \ldots, v_n)$ is related by some pbisim, using pre-calculated pbisims between nodes of the LTSs. It chooses the lexicographically least.

We explain how to calculate a node $v' = (v'_1, \ldots, v'_n)$ to which v is related, as determined by a particular σ-bisimulation p. The value v'_j at position j of tuple v' is determined as follows: find the leaf number, i, of the *Impl* leaf LTS such that p relates $LTS(i)$ nodes to $LTS(j)$ nodes, and set v'_j to the node of LTS(j) to which node v_i of $LTS(i)$ is related by p. Now, v represents $Pv_1 \parallel \ldots \parallel Pv_n$ where each Pv_i is the process represented by node v_i of $LTS(i)$, and by construction each Pv_i is such that $Pv_i = Pv'_j\sigma$, for some distinct index j (by uniformity of p). So, $Pv_1 \parallel \ldots \parallel Pv_n = Pv'_1\sigma \parallel \ldots \parallel Pv'_n\sigma = (Pv'_1 \parallel \ldots \parallel Pv'_n)\sigma$ and hence $Pv = Pv'\sigma$.

For improved efficiency, our implementation pre-calculates, for each pbisim, the appropriate ordering of indices i to calculate the components of v' in left-to-right order. It abandons calculation of v' when a component v'_j is calculated that makes the partial v' larger than the lex-least thus far.

When using *genRep*, before exploration we transitively close the calculated pbisims in the obvious sense. So it suffices to find just a *generating set of pbisims* (i.e., pbisims for a generating set of permutations of G) using the rules in Table 1 or an extended table. Transitive closure is not used for *sortRep* since *sortRep* does not generate all related nodes – even partially – and can in fact be determined applicable given a small number of suitable generating pbisims.

7.2 SymCheck3

Define SymCheck3 to be TwistedCheck(*repPair*3) where:

$$repPair3(u, v) = (u\sigma, rep(v), \sigma), \text{ some } \sigma \text{ in } G \text{ s.t. } v\sigma = rep(v)$$
$$rep = genRep$$

So SymCheck3 explores the *Spec*-*Impl* product space by following *repPair*3-twisted paths. Theorem 2 applies directly to SymCheck3 when the *Spec* and *Impl* transition systems are each G-symmetric. We drop the condition (which was needed for SymCheck1 and SymCheck2) that the *Spec* transition system T_{Spec} is *universally* G-symmetric – this condition is not needed here because *repPair*3 yields $u\sigma$ in the first part of its result, exactly as needed for Theorem 2 to apply. Hence this algorithm is more general than SymCheck2; the price paid for this extra generality is the need to calculate $u\sigma$, but this is straightforward given pbisims for T_{Spec}. Note that it would not be appropriate to use $rep(u)$ instead of $u\sigma$ in the definition of *repPair*3, as these will be different in general.

8 Experimental Results

We present results obtained using a prototype tool written in Perl. The tool compiles given *Spec* and *Impl* processes, checks particular symmetries of them claimed by the user and in so doing finds corresponding pbisims. It then checks applicability of, and runs, refinement checking algorithms requested by the user. The results presented are for the usual refinement checking algorithm (called Check here) and for algorithms SymCheck1, SymCheck2 and SymCheck3.

Specification processes were chosen that are refined by the implementations, to show the full (product) state space sizes. Three classes of refinement check are reported, distinguished by the choice of specification and implementation:

- Refinement of RUN(*Events*) by Towers of Hanoi models with 4 discs and 4-7 poles, where RUN(*Events*) can always perform any event;
- Refinement of RUN($\{|$ *try*, *enter*, *leave* $|\}$), which can perform all events on channels *try*, *enter* and *leave*, by Dijkstra mutual exclusion algorithm models with 2-4 participants; and
- Refinement of SpecME by these Dijkstra models, where SpecME can perform exactly the desired patterns of $try.i$, $enter.i$ and $leave.i$ events and is not universally symmetric for any non-trivial permutation.

Table 2. Experimental results for the usual refinement checking algorithm Check and the three SymCheck algorithms

Algorithm	Spec	Impl	States		Time (secs)					Number of perm syms found	Overhead per state explored
					Impl compilation + sym checking	Impl super-compilation	Impl pbisim transitive closure	Exploration			
Check	RUN {| Events |}	hanoi4p4d	256	100%	3.34	0.20	-	3.58	100%	-	0%
SymCheck1		hanoi4p4d	51	19.92%	3.34	0.20	-	0.73	20%	6	2%
SymCheck2		hanoi4p4d	51	19.92%	3.34	0.20	0.01	0.73	20%	6	2%
Check		hanoi5p4d	625	100%	5.13	0.37	-	19.14	100%	-	0%
SymCheck1		hanoi5p4d	52	8.32%	5.13	0.37	-	1.53	8%	24	-4%
SymCheck2		hanoi5p4d	52	8.32%	5.13	0.37	0.05	1.72	9%	24	8%
Check		hanoi6p4d	1296	100%	7.38	0.78	-	82.29	100%	-	0%
SymCheck1		hanoi6p4d	52	4.01%	7.38	0.78	-	3.11	4%	120	-6%
SymCheck2		hanoi6p4d	52	4.01%	7.38	0.78	0.56	4.20	5%	120	27%
Check		hanoi7p4d	2401	100%	9.88	3.78	-	246.83	100%	-	0%
SymCheck1		hanoi7p4d	52	2.17%	9.88	3.78	-	5.09	2%	720	-5%
SymCheck2		hanoi7p4d	52	2.17%	9.88	3.78	5.69	15.38	6%	720	188%
Check	RUN {| try,enter,leave |}	DijkstraME_2	445	100%	4.18	0.53	-	0.16	100%	-	0%
SymCheck2		DijkstraME_2	224	50.34%	4.18	0.53	0.00	0.11	69%	2	37%
Check		DijkstraME_3	19161	100%	22.66	2.65	-	11.77	100%	-	0%
SymCheck2		DijkstraME_3	3269	17.06%	22.66	2.65	0.01	2.94	25%	6	46%
Check		DijkstraME_4	1189379	100%	63.89	10.02	-	1103.00	100%	-	0%
SymCheck2		DijkstraME_4	51571	4.34%	63.89	10.02	0.07	118.95	11%	24	149%
Check	SpecME	DijkstraME_2	445	100%	4.08	0.52	-	0.13	100%	-	0%
SymCheck3		DijkstraME_2	224	50.34%	4.08	0.52	0.00	0.12	92%	2	83%
Check		DijkstraME_3	19161	100%	22.77	2.66	-	8.82	100%	-	0%
SymCheck3		DijkstraME_3	3269	17.06%	22.77	2.66	0.01	2.96	34%	6	97%
Check		DijkstraME_4	1189379	100%	63.57	10.00	-	861.12	100%	-	0%
SymCheck3		DijkstraME_4	51571	4.34%	63.57	10.00	0.07	133.88	16%	24	259%

Table 2 shows the results obtained for the most efficient of the applicable symmetry exploiting algorithms. For each check, G is the full symmetry group on pole indices (except pole A, where all discs start) or participant identifiers.

In each case the applicable SymCheck algorithms can be determined automatically based on whether there is found to be a full set of simple swap pbisims (in which case *sortRep* can be used) and whether the specification process LTS is found to be universally G-symmetric (in which case SymCheck2 applies, and so does SymCheck1 if *sortRep* can be used).

One column gives total time for compilation of the implementation process to a structured machine plus checking of the claimed implementation symmetries. Others give supercompilation time,[3] and time for transitive closure of implementation transition system pbisims (i.e., for determining an implementation transition system pbisim for each permutation in G, which is needed for *genRep* and so for SymCheck2 and SymCheck3). Corresponding timings are omitted for the specification as they are much smaller. Exploration times are reported.

Although the table does not show it, SymCheck3 has a larger overhead per state explored than does SymCheck2. The table does however include evidence that SymCheck2 has a larger overhead than SymCheck1.

[3] Supercompilation [6] can reduce exploration times greatly, but is outside the scope of this paper.

The Towers of Hanoi models are very simple. Each has a structured machine with a full set of simple swap pbisims for G. Also, the specification RUN($Events$) is universally G-symmetric. These properties are determined quickly by the tool and hence SymCheck1 is found to apply. Compared with Check, there is a substantial reduction in the number of state pairs explored by SymCheck1 and in exploration time. Although the compilation effort is larger, the extra costs are evidently small compared to the benefits of exploring fewer state pairs.

The Dijkstra mutual exclusion models were chosen in part because their structured machines (not shown) happen not to have simple swap pbisims for the permutations in the corresponding group G. Accordingly, SymCheck1 does not apply to these models. In contrast, SymCheck2 does apply when checking refinement of RUN($\{|\ try, enter, leave\ |\}$), as this specification is univerally G-symmetric. However, SymCheck2 does not apply with the just G-symmetric specification SpecME; only SymCheck3 applies with this refinement property.

For the larger symmetry groups, algorithms SymCheck2 and SymCheck3 suffer from the rapid increases in the size of G that result from increasing the number of poles or participants; this is because both algorithms use *genRep*, which needs a pbisim for each element of G. SymCheck1 is much less sensitive to this because it uses *sortRep*, which only requires existence of a linear number of (simple swap) pbisims. Furthermore, confirmation that these pbisims exist can be performed efficiently by checking existence of two pbisims that correspond to any transposition $(x\ y)$ and any cycle on all elements of G except for x. This method was used in our implementation of SymCheck1.

9 Conclusions

We have successfully adapted the representative function symmetry exploitation approach from the temporal logic model checking context to CSP refinement checking. The major obstacle was the need for access to sufficient local information about state during refinement checking, which is provided by representing the implementation process as a structured machine. We have also presented two generalisations of the basic algorithm. All three algorithms have been presented in a common style, in terms of a curried function TwistedCheck.

An option for future work is to characterise more precisely, in terms of processes, when alternative SymCheck variants apply and perhaps develop methods for transforming CSP models, or their transition systems, to make the more efficient algorithms more widely applicable.

There are many other possible extensions, including: use of (a perhaps liberalized notion of) data independence to increase the efficiency of symmetry identification; development of variants of the *sortRep* function to cope efficiently with wider classes of structured machines and permutation bisimulations over them (and hence more implementation processes); extension to multiple representatives; extension to virtual symmetries [12]; and use of computational group theory to improve efficiency.

It would also be interesting to investigate the temporal logic analogue of (non-universal) G-symmetry and perhaps generalise the representative function approach to symmetry exploitation for temporal logic model checking, effectively removing the requirement that the specification is always symmetric.

The experimental results are encouraging. They illustrate that the refinement checking algorithms presented can give significant savings in the number of state pairs explored and in verification time. The former can be expected to lead to corresponding reductions in memory usage, which is often the dominant factor determining the sizes of problems that can be checked.

Acknowledgements

We are grateful to the reviewers for their valuable comments.

References

1. Miller, A., Donaldson, A.F., Calder, M.: Symmetry in Temporal Logic Model Checking. ACM Comput. Surv. 38(3) (2006)
2. Hoare, C.A.R.: Communicating Sequential Processes. CACM, 21(8) (1978)
3. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
4. Formal Systems (Europe) Ltd: Failures-Divergences Refinement: FDR2 User Manual (1992-2008)
5. Moffat, N., Goldsmith, M., Roscoe, A.W.: Towards Symmetry Aware Refinement Checking (Extended Abstract). In: Proceedings of International Symmetry Conference, Edinburgh, UK (2007)
6. Ryan, P., Schneider, S., Goldsmith, M., Lowe, G., Roscoe, A.W.: Modelling and Analysis of Security Protocols. Addison-Wesley, Reading (2001)
7. Park, D.M.: Concurrency on automata and infinite sequences. In: Deussen, P. (ed.) GI-TCS 1981. LNCS, vol. 104. Springer, Heidelberg (1981)
8. Milner, R.: Communication and concurrency. Prentice-Hall, Englewood Cliffs (1989)
9. Clarke, E., Enders, R., Filkhorn, T., Jha, S.: Exploiting Symmetry in Temporal Logic Model Checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
10. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design 9(12), 105–131 (1996)
11. Lazić, R.S.: A semantic study of data-independence with applications to the mechanical verification of concurrent systems. Ph.D. thesis, Oxford University Computing Laboratory (1999)
12. Emerson, E.A., Havlicek, J., Trefler, R.: Virtual Symmetry Reduction. In: Proceedings of the 15th IEEE Symposium on Logic in Computer Science (2000)

Appendix: Theory

Lemma 1. *Let G be a group of event permutations. Consider the product space for particular Spec and Impl transition systems, with initial state pair $s_0 =$*

(u_0, v_0). *Suppose repPair is a function that maps each state pair* (u, v) *to* $(u\sigma_{u,v}, v\sigma_{u,v}, \sigma_{u,v})$, *some* $\sigma_{u,v}$ *in* G. *Then, for all traces* t, *there is a path* p *from* s_0 *to state pair* $s = (u, v)$, *with* $recover(p) = t$ *iff there is a repPair-twisted path* r *from* s_0 *to* $s\sigma = (u\sigma, v\sigma)$, *with* $recover2(r) = t\sigma$, *some* σ *in* G.

Proof. Induction on length k of t.
Base case: $k = 0$, so $t = \langle\rangle$. There is exactly one path, $p = \langle s_0\rangle$, starting at s_0 and such that $recover(p) = \langle\rangle$. This path ends at $(u, v) = (u_0, v_0) = s_0$. Also, there is exactly one *repPair*-twisted path, $r = \langle s_0\rangle$, starting at s_0 and such that $recover2(r) = \langle\rangle$. This *repPair*-twisted path ends at $s_0 = s_0 1$ so we may choose $\sigma = 1$. Then $recover2(r) = \langle\rangle = \langle\rangle\sigma$.

Inductive step: Suppose the lemma holds for all traces t of some length k. We show it also holds for all t of length $k + 1$.
($\Rightarrow$) Suppose $p = p'^\frown\langle e\rangle$ is a length-k+1 path from s_0 to (u, v) and $recover(p) = t$. Then p' is a length-k path to some state pair (u', v') and there is a transition $(u', v') \xrightarrow{e} (u, v)$. Clearly, $recover(p) = recover(p'^\frown\langle e\rangle) = recover(p')^\frown\langle e\rangle$. So, defining $t' = recover(p')$, we have $t = t'^\frown\langle e\rangle$. By the induction hypothesis applied to p' and t', there is a *repPair*-twisted path r' from s_0 to $(u'\sigma', v'\sigma')$ with $recover2(r') = t'\sigma'$, some σ' in G. Recall there is a transition $(u', v') \xrightarrow{e} (u, v)$, so there is a transition $(u'\sigma', v'\sigma') \xrightarrow{e\sigma'} (u\sigma', v\sigma')$. Let ρ in G be such that $repPair(u\sigma', v\sigma') = (u\sigma'\rho, v\sigma'\rho, \rho)$.[4] Then $r = r'^\frown\langle e\sigma', \rho, (u\sigma'\rho, v\sigma'\rho)\rangle$ is a *repPair*-twisted path to $(u\sigma'\rho, v\sigma'\rho)$ since r' is *repPair*-twisted and ends at $(u'\sigma', v'\sigma')$ and there is a transition $(u'\sigma', v'\sigma') \xrightarrow{e\sigma'} (u\sigma', v\sigma')$ and $repPair(u\sigma', v\sigma') = (u\sigma'\rho, v\sigma'\rho, \rho)$. Putting $\sigma = \sigma'\rho$, we obtain that r is a *repPair*-twisted path from s_0 to $s\sigma = (u\sigma, v\sigma)$, where $\sigma = \sigma'\rho$ is in G since both σ' and ρ are. It remains to show that $recover2(r) = t\sigma$. We have $recover2(r) = recover2(r'^\frown\langle e\sigma', \rho, (u\sigma'\rho, v\sigma'\rho)\rangle) = (recover2(r')^\frown\langle e\sigma'\rangle)\rho = (t'\sigma'^\frown\langle e\sigma'\rangle)\rho = (t'^\frown\langle e\rangle)\sigma'\rho = t\sigma$.
($\Leftarrow$) Similar.

Theorem 1. *Let* G *be a group of event permutations and suppose Spec and Impl have transition systems* T_{Spec} *and* T_{Impl} *respectively. Suppose function repPair maps each state pair* (u, v) *to* $(u\sigma_{u,v}, v\sigma_{u,v}, \sigma_{u,v})$ *for some* $\sigma_{u,v}$ *in* G. *Then Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl has a counterexample trace* t *iff* $\exists\sigma \in G$ *s.t. Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl has a repPair counterexample trace* $t\sigma$.

Proof. ($\Rightarrow$) Let t be a counterexample trace of *Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl*. Then t is a trace of *Impl*. Let t_1 be the longest prefix of t that is a trace of *Spec* and t_2 be such that $t = t_1^\frown t_2$. Then there is a path p from initial state pair s_0 to $s = (u, v)$, say, with $recover(p) = t_1$ and t_2 a bad trace from (u, v). By Lemma 1, there is a *repPair*-twisted path r from s_0 to $s\sigma$, some σ in G, with $recover2(r) = t_1\sigma$. But $s\sigma = (u\sigma, v\sigma)$ is a bad state pair, and $t_2\sigma$ must be a bad trace from $s\sigma$ (since *Impl* state $v\sigma$ is able to perform trace $t_2\sigma$ but *Spec* state $u\sigma$ is not). So $recover2(r)^\frown t_2\sigma = t_1\sigma^\frown t_2\sigma = (t_1^\frown t_2)\sigma = t\sigma$ is a *repPair* counterexample trace, for this σ in G.

[4] Such a ρ is denoted $\sigma_{u\sigma', v\sigma'}$ in the statement of the lemma.

($\Leftarrow$) Let t be a *repPair* counterexample trace of *Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl*. Then $t = recover2(r)^\frown t_2$ for some *repPair*-twisted path r from initial state pair s_0 to a bad state pair $s = (u, v)$, such that t_2 is a bad trace from (u, v). So *Impl* state v can perform trace t_2 but *Spec* state u cannot. Putting $t_1 = recover2(r)$, we have $t = t_1{}^\frown t_2$. By Lemma 1, there is a path p from s_0 to $s\sigma^{-1}$, some σ^{-1} in G, with $recover(p) = t_1\sigma^{-1}$. Now $s\sigma^{-1} = (u\sigma^{-1}, v\sigma^{-1})$ must be a bad state pair with $t_2\sigma^{-1}$ a bad trace from $s\sigma^{-1}$, since *Impl* state $v\sigma^{-1}$ can perform trace $t_2\sigma^{-1}$ but *Spec* state $u\sigma^{-1}$ cannot. So p is a path from s_0 to bad state pair $s\sigma^{-1}$ and $recover(p)^\frown t_2\sigma^{-1} = t_1\sigma^{-1}{}^\frown t_2\sigma^{-1} = (t_1{}^\frown t_2)\sigma^{-1} = t\sigma^{-1}$ is a counterexample trace, for this σ^{-1} in G.

Theorem 2. *Let G be a group of event permutations and suppose Spec and Impl have G-symmetric transition systems T_{Spec} and T_{Impl} respectively. Suppose function repPair maps each state pair (u, v) to $(u\sigma_{u,v}, v\sigma_{u,v}, \sigma_{u,v})$ for some $\sigma_{u,v}$ in G. Then Spec $\sqsubseteq_{\mathcal{T}}$ Impl has a counterexample trace t iff Spec $\sqsubseteq_{\mathcal{T}}$ Impl has a repPair counterexample trace t.*

Proof. By Theorem 1, *Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl* has a counterexample trace t iff $\exists\, \sigma \in G$ s.t. *Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl* has a *repPair* counterexample trace $t\sigma$. Then use that, $\forall\, \sigma$ in G, *Spec* $\sqsubseteq_{\mathcal{T}}$ *Impl* has a counterexample trace t iff it has a counterexample trace $t\sigma$ (which follows easily from G-symmetry of the *Spec* and *Impl* transition systems).

Probing the Depths of CSP-M: A New FDR-Compliant Validation Tool

Michael Leuschel and Marc Fontaine

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

Abstract. We present a new animation and model checking tool for CSP. The tool covers the CSP-M language, as supported by existing tools such as FDR and PROBE. Compared to those tools, it provides visual feedback in the source code, has an LTL model checker and can be used for combined CSP ‖ B specifications. During the development of the tool some intricate issues were uncovered with the CSP-M language. We discuss those issues, and provide suggestions for improvement. We also explain how we have ensured conformance with FDR, by using FDR itself to validate our tool's output. We also provide empirical evidence on the performance of our tool compared to FDR, showing that it can be used on industrial-strength specifications.

Keywords: CSP, Tool Support, Model Checking, Animation, B-Method, Integrated Formal Methods, Specification Language Design, Logic Programming.

1 Introduction

CSP and B can be effectively used together in a complementary way to formally specify systems [29,3,5]. B can be used to specify abstract state and can be used to specify operations of a system in terms of their enabling conditions and effect on the abstract state. CSP can be used to give an overall specification of the coordination of operations.

The overall goal of this research project was to obtain an animation and model checking tool which could be used to validate such combined B and CSP specifications. A major requirement was that this tool should deal with full B specifications in AMN (abstract machine notation) as well as with full CSP specifications in CSP-M (machine readable CSP), so that existing industrial tools such as Atelier B [28] or FDR [7], could be applied to validate the individual B or CSP specifications in isolation.

* A substantial part of this research has been sponsored by AWE, plc within the project "ProCSB" as well as by the EU funded FP7 project 214158: DEPLOY (Industrial deployment of advanced system engineering methods for high productivity and dependability).

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 278–297, 2008.

After previous attempts with other tools,[1] in this project it was decided to extend the ProB tool [16], in order to deal with CSP-M specifications. Indeed, ProB already deals with full AMN and provides both animation and model checking facilities. ProB can also be applied with reasonable effort to new specification languages, provided a Prolog interpreter for that language is available. In earlier work, ProB was already extended to validate combined B and CSP specifications [5]. However, this work was based on the earlier interpreter [15], which only treats a very small subset of CSP-M (e.g., there are no replicated operators, no channel declarations, no let declarations or lambda constructs, etc.), and used its own incompatible syntax (which was designed at the time to enable easier parsing by Prolog). In other words, the previous tool was an academic prototype;[2] what is required is a new tool which can handle existing real-life CSP-M specifications.

A major hurdle of this research project was the development of a new parser and interpreter for full CSP-M for use within ProB. The issue of synchronising the B interpreter with the CSP interpreter is mainly achieved in the same manner — via Prolog unification — as in the earlier work [5]. Hence, in the rest of this paper, we will only briefly discuss the issue of synchronising the B interpreter with the CSP interpreter and concentrate on presenting the new CSP-M parser, type checker, and interpreter. This is also warranted, since we have actually also gained a new powerful tool to animate and model check CSP-M specifications. This tool overcomes some of the drawbacks of existing tools FDR and PROBE, and adds several new features:

- precise syntax error highlighting in the source code
- precise semantic error highlighting in the source code
- visual feedback on complicated events in the source code
- the ability to deal with infinite state sub-processes and to some extent also main processes
- graphical visualisation of the state space of CSP-M specifications
- the ability to apply LTL model checking to CSP-M [20]
- a new type checker for CSP-M [6]
- a web-interface using Google-Web-Toolkit (`www.myprob.de`)

In the remainder of this paper you will find various critiques of CSP-M, and also of FDR and PROBE. To clarify our position, we believe that FDR and PROBE are two very useful tools, and that CSP-M is a very attractive specification language. There are also aspects of CSP that are dealt with much better by FDR and PROBE, than by our tool (e.g., deeply nested processes or compression [25]). Exactly because we believe that CSP-M and the existing tools are relevant, are we interested

[1] E.g., combining the commercial BToolkit animator from BCore with the commercial PROBE animator [8] from Formal Systems Ltd. in the EPSRC funded project GR/R96859/01 "Verification and Development of Interacting Software Components".

[2] Indeed, our new interpreter alone is roughly six times the size of the interpreter in [5].

in improving and clarifying the language. We hope that we have provided insights and a new tool of value to the formal engineering methods community beyond CSP || B.

Outline. We start off be outlining the difficulties of treating CSP-M in Section 2, also pinpointing some semantic issues of the language, such as lack of substitutivity of equals. We describe our Prolog interpreter in more detail in Section 3 and the new parser in Section 4. In Section 5 we discuss how we have ensured compliance of our tool with the existing CSP-M tools. In Section 6 we present empirical results of using our new tool. We conclude with discussions and more related work in Section 7.

2 Challenges of Validating CSP-M

While core CSP [10] is a relatively succinct language, full CSP-M as supported by FDR is a very extensive and expressive specification language. Some of the aspects that make CSP-M a challenging target for a formal validation tool are:

- CSP-M is a dynamically typed language.
- CSP-M comprises a higher-order functional programming language.
- CSP-M contains an extensive range of datatypes: booleans, integers, tuples, associative tuples, sequences, sets, and combinations thereof. The use of an associative datatype is rather unique, and enables some elegant formalisations. On the other hand it considerably complicates the life of the tool developer, as we show below.
- CSP-M has grown and matured over a considerable period of time, and many features were added, e.g.:
  - Complex pattern matching, even allowing combinations of patterns to be expressed using the `@@` operator. One unique aspect of CSP-M is the ability to use the concatenation operator `^` inside function patterns. E.g., one can define a function that computes the last element of a list by: `last(s^<x>) = x` (no mainstream functional or logic programming language allows this).
  - Complex channel matching with associative tuples and trailing question marks which can match tuples of arbitrary length.
  - Closure operations on partially constructed datavalues; partially constructed datavalues can be passed to functions.
  - Nested function definitions, which can appear in many places (e.g., inside an individual channel output expression)

The complexity of CSP-M is also reflected in its syntax, which makes parsing surprisingly hard (see Section 4). As we will show below, some parts of CSP-M also have an overly complicated semantics, which is furthermore formalised neither in Roscoe's book [24] nor in the FDR manual. We will discuss some of the subtle, problematic points in the remainder of this section. How our implementation copes with higher-order functions and nested functions is explained in Section 3.

Dynamic Typing. CSP-M, as accepted by FDR and PROBE [8], is a dynamically typed language. Indeed, while channels must be statically typed by the user with channel declarations, CSP-M provides no way of declaring types for functions and variables. Moreover, the existing tools FDR and PROBE accept specifications (without warning or runtime error message) in which there is no way to statically type variables or functions, as the following example shows:

```
datatype COL = red | green
channel ci:{0..4}
channel cc:COL
BUF(<>) = STOP
BUF(<n,val>^s) = ci!n -> if n<2 then ci!val->BUF(s) else cc!val->BUF(s)
MAIN = BUF(<1,1,3,red,1,4>)
```

Here the argument to BUF is a sequence consisting of mixed integers and colours, the type of every second item depending on the value of the preceding integer. Hence, at least as far as the existing tools FDR and PROBE are concerned, CSP-M is a dynamically typed language. This means, e.g., that an interpreter cannot rely that operation arguments (e.g., for arithmetic operations) have the correct type and must perform type checks at runtime.

There now is a prototype static type checker available for download from Formal Systems. It would reject the above specification; but it will also sometimes reject legal specs (see further below). For critical applications, we believe that it is good practice to write only specifications that are accepted by such a static type checker, even if the tools FDR and PROBE can deal with dynamically typed specifications. Still, in order to be compliant with these existing tools, we also support dynamic typing, but we have also developed an (optional) static type checker [6].

The Trouble with Tuples. Associative tuples can be constructed using the dot (.) operator. This, however, is not the only use of the dot operator and actual meaning depends on the context of use and on prior datatype declarations. Take, e.g., the following CSP-M specification:

```
channel a,a': {0..9}.{0..9}   {- Meaning 1. -}
datatype R = r.{0..9}         {- Meaning 2. -}
channel b,b':R
MAIN = Rep1(a,a',2.3);        {- Meaning 3. -}
       Rep2(b,b',r.3)         {- Meaning 4. -}
Rep1(c,c',x) = c!x -> SKIP [| {|c|} |] c?y?z -> c'!y!z -> SKIP
Rep2(c,c',x) = c!x -> SKIP [| {|c|} |] c?y?z -> c'!y -> SKIP
```

This example (which is type correct according to Formal Systems type checker) shows that the dot has actually four different meanings:

1. Here the dot generates a cartesian product of two sets, and not a tuple of two sets. This is made clear that if one defines `T = {0..9}.{0..9}` and then tries to declare `channel a,a':T` one gets the FDR error message "Value

{0,1,2,3,4,5,6,7,8,9}.{0,1,2,3,4,5,6,7,8,9} is not a set." Thus, inside a channel declaration the dot acts as a Cartesian product, and substitutivity of equals does not hold: one *cannot* replace {0..9}.{0..9} by T even though they are declared to be equal. One could hence argue that CSP-M is not referentially transparent.

2. Here we construct a set of records. This is similar to point 1, but this time the first argument is not a set but a constructor. This use of the dot occurs inside a datatype declaration.
3. Here the dot operator constructs an associative tuple, which can be deconstructed into its two constituents using the c?y?z channel prefix in Rep1.
4. In contrast to point 3, here we do *not* construct a tuple, but a record. The difference becomes apparent in the channel prefix c?y?z in Rep2, where this time y gets bound to the entire record and z is bound to the empty tuple. Hence the output c'!y is *not* a type error and is accepted by FDR. Note that calling Rep2(a,a',2.3) would lead to a type error.[3]

Meanings 3. and 4. occur outside of channel and datatype declarations, and they can be distinguished by checking whether the first component of a dot is a record constructor declared in a datatype declaration. Also note that there is no way to statically distinguish between case 3 and 4, as a slight adaptation of the above example shows:

```
Rep3(c,c',x1,x2) = c!x1.x2 -> SKIP [| {|c|} |] c?y?z -> c'!y!z -> SKIP
MAIN2 = Rep3(a,a',2,3); Rep3(b,b',r,3)
```

This time the occurrence of the dot operator inside Rep3 will, depending on its arguments, either construct a tuple (for the call Rep3(a,a',2,3)) or a record (for the call Rep3(b,b',r,3)).

We have faithfully implemented these various uses of dot, checking conformity with FDR and PROBE (see Section 5). However, we believe that this aspect of CSP-M should be simplified in future, at least using another operator for record construction. Indeed, even to several CSP-experts (some authors of books about CSP), the difference in behaviour between point 3 and 4 came as a surprise. Only after experimenting with FDR and PROBE and discussions with Michael Goldsmith (one of the main developers and researchers behind FDR) did we realise that there was a subtle, but important difference between the third and fourth use of the dot operator. We think that such semantic pitfalls should be avoided in languages aimed at formal modelling of critical systems.

As a side-note, the empty tuple to which z is bound inside Rep2 does have no syntactic denotation in CSP-M. We also believe that this should be remedied in future revisions, as it prevents certain source-to-source transformations to be applied to CSP-M specifications (e.g., constant folding cannot be applied if the constant happens to be the empty tuple).

[3] More precisely, FDR says "Mismatch calculating augment processing communication !y near line 9 of ThreeDots.csp on channel a' for event a'.8.9".

Associativity of Tuples. The associativity of tuples poses some major headaches, both for static analysis of CSP-M specifications, as well as for animation and model checking tools. For example, given a prefix `c3!x?val!y -> ...` and a channel declaration `channel c3:{0..4}.COL.{0..4}` one would assume that it is possible to statically infer the type of `val` to be of type `COL`. As the following example shows, this is not the case:

```
datatype COL = red | green
channel a:{0..4}
channel c3:{0..4}.COL.{0..4}
MAIN = a?x?e -> if x<2 then P(x.red,e) else P(x,x)
P(x,y) = c3!x?val!y -> STOP  -- here val can be either Int or COL
```

Let us examine the `MAIN` process after `a?x?e`: the variable `x` will contain a number between 0 and 4 and `e` will always contain the empty tuple. If the number is smaller than 2, `val` inside the `P` process will be of type integer and not of type colour. If `x` is greater or equal to 2, then `val` will be of type `COL`.

This means that channel prefixing is a surprisingly complex operation in CSP-M. In addition to mystifying users, the above also makes the life of the tool developers much harder. For example, the result is that the Formal Systems type checker sometimes fails to type-check a program, even if it can in principle be statically typed. More concretely, if `MAIN` is replaced by `MAIN = a?x?e -> P(x.red,e)` then `val` is always of type integer, but the type checker fails to type-check the specification.

We also found out that in some circumstances FDR and PROBE did not fully implement all consequences of the associativity of the tuples. Indeed, changing the definition of `P` above into the following one, leads to an "unsupported comparison processing communication error" in both FDR and PROBE:

```
channel b:COL
P(x,y) = c3!x?val!y -> if x<2 then a!val -> STOP else b!val -> STOP
```

Our tool does support all of the above specifications. Still, as in the case of the different meanings of the dot operator above, we believe that this aspect of CSP-M should be cleaned up in future versions. The additional expressiveness incurred by associativity and flexible channel matching is in our opinion not sufficient to counterbalance the lack of predictability and clarity for the user.

3 The CSP-M Prolog Interpreter

In earlier work [5], synchronisation of B and CSP was achieved by

- having interpreters for B and CSP both in Prolog and
- using Prolog unification to synchronise the two interpreters.

This provided an animator and model checker for CSP ‖ B specifications.

This aproach has proven to work very well and has been kept in this project. Figure 1 shows the general architecture of our development. The "Sync" box

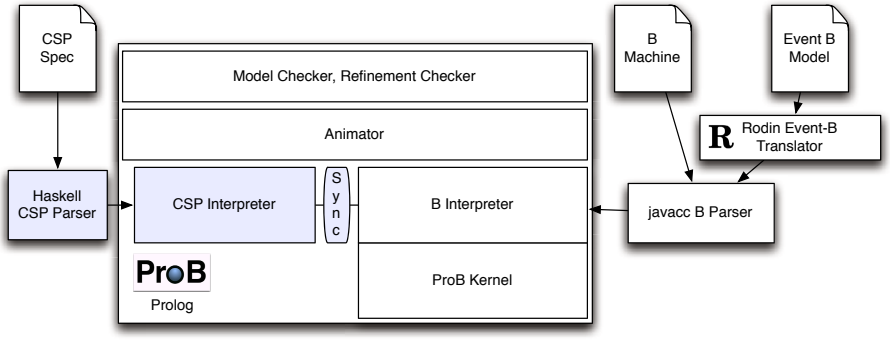


Fig. 1. Architecture

required only relatively minor changes to the existing code base from [5]. The CSP-Parser and Interpreter, however, have been completely re-developed. As already mentioned, the previous parser was incompatible with FDR and the previous interpreter only supported the core CSP language, leaving many of the difficult aspects of CSP on the side, notably the ones detailed in Section 2. This required the development of several intricate compilation techniques and explains the sixfold increase in code size of our new interpreter.

We describe the new parser in Section 4. The new CSP-M interpreter was inspired by the earlier interpreter in [5], which in turn was inspired by [15]. Just as in [5], co-routining was used to enable us to write a compositional interpreter, and translating the operational semantics rules of Roscoe [24]. The Prolog code implements a ternary relation *cspm_trans*, where $cspm\_trans(e, a, e')$ means that the CSP-M expression e can evolve into the expression e' by performing the event a.

Why co-routining? In classical Prolog literals within a goal are selected strictly from left to right. Co-routines (sometimes also called delay or wait declarations) [22] enable a programmer to influence Prolog's selection rule via `when`, `block` or `freeze` declarations. For example, take the following two clauses, defining list membership:

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

Now take the query goal `?- member(X,L), L=[1], X=2.` In classical Prolog, the leftmost literal `member(X,L)` will be selected first. As `member(X,L)` has infinitely many solutions, the above query will not terminate and the Prolog interpreter will never realise that there is no solution for the query. By adding, e.g., a block declaration `:- block member(?,-).` , we ensure that all calls to `member` will delay until the second argument is not a variable anymore. For the above goal, Prolog will thus suspend the call `member(X,L)`, and execute `L=[1]` first. This will awaken `member(X,L)`, which now finds only a single solution `X=1`, after which the call `X=2` simply fails.

For a pure logic program, these declarations do not modify the logical meaning, but they can be used to improve the runtime behaviour of the program. In particular, one can ensure that an interpreter for CSP will terminate, or be much more efficient, in circumstances where Prolog's classical left-to-right selection rule would lead to non-termination or to unacceptable performance.

Take the following example (part of the regression test for our new tool):

```
MyInt = {0..99}
channel ch: MyInt.MyInt
channel out,out2: MyInt
T(x) = ch!x?y:{z|z<-MyInt, z<x} -> out!y -> T(y) []
       ch!x?y:{z|z<-MyInt, z>=x} -> out2!y -> T(y)
S(x) = ch?y!x -> S(y)
MAIN =  T(1) [| {| ch |} |] S(2)
```

Let us examine how the interpreter handles the synchronisation operator `[|{|ch|}|]` by looking at one of the inference rules from [24]:

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P \mid\mid_X Q \xrightarrow{a} P' \mid\mid_X Q'} \quad (a \in X)$$

Here $P = T(1)$, $Q = S(2)$ and X is the closure of ch, i.e., $X = \{ch.0.0, ch.0.1, \ldots, ch.99.99\}$. In order to determine whether this rule is applicable, the interpreter will first compute the outgoing transitions of $T(1)$. One problematic issue here is that the interpreter does not yet know the value of y and hence cannot yet determine whether y is a member of the set comprehensions inside T's definition. One solution would be to simply enumerate all possible values for y, as indicated by the type of the channel.[4] This is actually the approach employed by FDR and PROBE. The drawback, however, is that a lot of effort can be wasted if the type of the channel value is large (e.g., here T(1) has 100 possible events), and there is no way to treat channels with unbounded types.

We have overcome this limitation by employing co-routining: basically the value of y will be left uninstantiated (a free Prolog variable) and the membership test delays until the value is known. As it is possible that some channel fields are never given an explicit value, the interpreter is wrapped into an outer layer which will enumerate any remaining uninstantiated channel fields at the very end.

Several other constructs will delay until their arguments are sufficiently instantiated. This has efficiency advantages, but also allows one to translate the logical inference rules from [24] in a relatively straightforward way, without having to worry in which order to evaluate the subexpressions nor having to perform additional analyses. The statespace of the above specification, as computed by our tool, is shown in the left of Fig. 2.

Here is a more realistic example with unbounded channel types. The overall state space of MAIN is finite, and our model checker can (quickly) check the

[4] In general, there can be a mixture of inputs and outputs, and information can flow both ways.

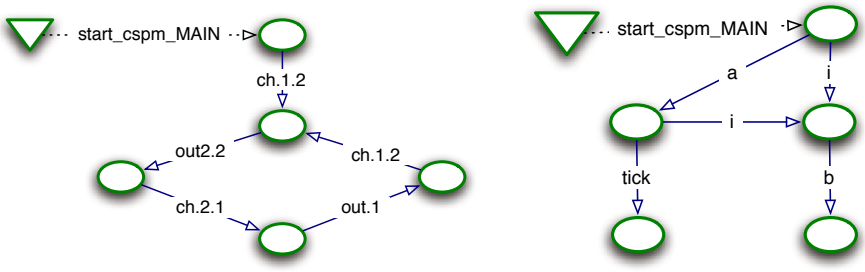


Fig. 2. Two statespaces as displayed by ProB

entire state space. The only trace of the system is $\langle out.1, out.1, out.2, out.3, out.5, gen.5\rangle$. Neither ProBE nor FDR can be used with this specification, because the channel types are not bounded.

```
channel out,gen: Int
FibGen(N,M) = if M<10000 then out!M -> FibGen(M,N+M) else STOP
Take(n) = if n>1 then out?_ -> Take(n-1)
                 else out?x -> gen!x -> STOP
FibSeq = FibGen(0,1)
MAIN = FibGen(0,1) [| {| out |} |] Take(5)
```

Our tool can also deal with truly infinite state processes, in the sense that they can be animated and partially validated. For example, our tool can find the deadlock in `MAIN = P [| {a,b} |] Q` where `P = (a -> P) [[ a<-b, b<- a]]` and `Q = a->Q [] b->a->b->STOP` after less than 0.01 s, whereas FDR goes into an infinite loop.

Very often a natural specification style is to have sub-processes which are infinite-state and only the global composition makes the system finite. For example, the following is a quite natural pattern, having an infinite state server process, but which is constrained by the environment so that the entire system (MAIN) is finite state and can be model checked. This specification can be exhaustively model checked using our tool, but FDR cannot deal with it because the tool tries to expand and normalise the infinite state `Server` process *before* conjoining it with the `User` process.

```
ID = {0..3}
channel new, ping, ack: ID
channel shutdown
Server =  new?id -> (Server ||| Serve(id)) [] shutdown -> STOP
Serve(id) = ping?id -> ack!id -> Serve(id)
User = new?i -> UserActive(i)
UserActive(i) = ping!i -> ack!i -> UserActive(i)
MAIN = Server [| {| new,ping,ack,shutdown |} |] User
```

Recursion and precompilation. Our tool stores the CSP-M functions in a precompiled Prolog database of definitions. It uses the so-called non-ground representation [9], meaning that variables in CSP-M functions are represented by

variables in the Prolog code. This enables us to use unification and Prolog argument indexing for efficient function lookup (see, e.g., the McCarthy benchmark in Section 6).

While the non-ground representation leads to an efficient interpreter, care has to be taken with local variables to ensure that different uses and unifications of the same local variable do not interfere with each other. A pre-compilation phase of our tool ensures that all local variables are pushed down into their own Prolog clauses, so that fresh copies are obtained before every use. Take for example the following CSP-M specification:

```
Repeat(X) = (X ; Repeat(X))
RepHalf = Repeat(out?x -> out!x/2 -> SKIP)
```

Here, the local variable `x` is "re-used" on every recursive call to `Repeat`. By generating an extra function (named here `RepHalf->_23`) for the prefix construct introducing the local variable, we ensure that every time a new copy of the local CSP-M variable is required, a fresh Prolog variable will be generated (as the function `RepHalf->_23` will be represented by an individual Prolog clause and every time the clause is used, fresh copies of the local variables are produced):

```
Repeat(X) = (X ; Repeat(X))
RepHalf = Repeat(RepHalf->__23)
RepHalf->__23 = out?x -> out!x/2 -> SKIP
```

Let expressions and lambda lifting. Nested let expressions are implemented using a static compilation technique called lambda lifting (or closure conversion) [13] (see also Chapter 13 of [14]). The idea is that every free variable of a nested function body is converted into an additional argument of the function, and the local function is replaced by a renamed global function. Take for example the following CSP-M specification:

```
Repeat(X) = (X ; Repeat(X))
RepCube(x) = Repeat(out!x -> (let f(m)=m*x  within out!f(m*x) -> SKIP))
```

We generate a new global function definition for the inner function `f`, adding the free variable `x` as extra argument of the function. The function is also given a special name (here `RepCube*f_1`), to avoid name clashes with user-defined functions and other lifted functions:

```
Repeat(X) = (X ; Repeat(X))
RepCube(x) = Repeat(out!x -> out! RepCube*f__1(x*x,x) -> SKIP)
RepCube*f__1(m,x) = m*x
```

Currying and Lambda Expressions. In addition to the standard syntax for functions, CSP-M also allows functions to be defined in curried form. In the following `f` is defined as a curried function and the `MAIN0` process should output 1.2 on the `out2` channel:

```
channel out2: {0..16}.{0..16}
f(x)(y) = out2!x!y -> SKIP
MAIN0 = Run(f(1))
Run(fc) = fc(2)
```

Curried function definitions are detected by our parser and our pre-compilation phase translates curried functions into lambda expressions. For example, internally the above function `f` is represented as if it was defined by the following:

```
f(x) = \y @ out2!x!y -> SKIP
```

This of course begs the question how CSP-M lambda abstractions have been implemented. First, the arguments of a lambda abstraction (i.e., the `y` in the example above) are frozen during a pre-compilation phase to ground Prolog terms (using the `numbervars` built-in, which instantiates all the free variables in a Prolog term to ground terms of the form `'$VAR'(`n`)`, where n is a number). When a lambda abstraction is applied, then its parameters are lifted again to real Prolog variables, which are then unified with the actual parameter values, after which the body of the lambda abstraction is evaluated. This scheme allows lambda abstractions to be applied multiple times with different parameter values.

Tracking Source Code Positions. Our parser was designed from the start to keep track of source code positions. Indeed, one aspect of FDR that can be frustrating, is that some of the error messages contain only approximate or no source code positions (e.g., for the error message "Function applied to a value outside its domain").

Our tool's source code positions are not only used to pinpoint syntax errors, but also to show to the user the exact location of certain "runtime" errors (e.g., when a process outputs a value that is outside of the channel type), as well as give feedback on events in the animator. Indeed, when animating the user can find out which locations in the source code contributed to a particular event (see Figure 3). This also works for τ events. This is achieved by the parser adding source code information inside the abstract syntax tree and by the interpreter keeping track of all source code positions that are involved in a certain event.

4 The New Parser in Haskell

As part of this work, we also implemented a new CSP-M parser. A major change from the existing FDR parser [26] is the incorporation of source code information inside the abstract syntax tree. Another design goal was that the code for the new parser should be understandable, and that it should be easy to extend and reuse the code in future projects. The reference implementation of the CSP-M parser [26] uses the bison parser generator, but the grammar, that actually serves as input for bison, is itself generated with a Perl script. This however, quite effectively obfuscates the parser and makes extending it very difficult.

Furthermore, apart from serving as input to our new CSP-M validation tool, we are also currently adding CSP-M support to the Eclipse IDE, as well as

```
Operate = (balance?a?v -> Operate
        [] transferReq?s?a1?a2?ok ->
              (transferExec!s!a1!a2 -> Operate
               [] abort -> Operate)
        [] logout -> mainB)
-- OZ Part Bank
-- We represent the current balance bal as a set of
-- pairs (account-id, value). This requires some
-- auxiliary functions defined below:
ValSet = \b,a @ { v | v <- Val, m((a,v),b) }
pick({x}) = x
PickVal = \b,a @ pick(ValSet(b,a))
withdrawOK = \b,a1,a2,s @
         not(a1==a2) and
         (PickVal(b,a1) - s >= 0)
upd = \b,a,v @
   let
         bminus = diff(b,{(a,vold) | vold <-Val })
   within
         union(bminus, {(a,v)})
-- The set of customers is defined as a concrete
-- subset of UserID. It appears as a global parameter
-- of the process OZB.
cust = {u1, u2}
OZB(bal,transferOK) =
     (m(bal,Set({(a,v)|a<-AccID,v<-Val})) and
      m(transferOK,Bool)) &
         (
         ([] (u,ok): {(u,m(u,cust)) | u <- UserID } @
              login.u.ok -> OZB(bal,transferOK))
         []
          ([] (a,v) :
           {(a,PickVal(bal,a)) |
              a <- AccID, card(ValSet(bal,a))==1 } @
           balance.a.v -> OZB(bal,transferOK))
        []
         ([] (s,a1,a2,ok):
         {(s,a1,a2,withdrawOK(bal,a1,a2,s)) |
            s<-Sum, a1<-AccID, a2 <-AccID,
           card(ValSet(bal,a1)) == 1 } @
          transferReq.s.a1.a2.ok -> OZB(bal,ok))
        [] transferExec?s?a1?a2 ->
```

Fig. 3. Source code highlighting the bank model from [2] (partial view)

producing new Haskell based tools for type checking CSP-M. To allow this interoperability, our parser can deliver parse-results in different output formats, e.g., as a set Prolog facts or as a Java object representing the abstract syntax tree. Our parser was developed with the Glasgow Haskell Compiler (GHC) and is currently available as dynamic-link-library for four different architectures. No knowledge of Haskell is required to use the parser.

Internally the parser is based on the Parsec combinator parser library [11]. Our practical experience, implementing a combinator parser for CSP-M in Haskell, was overall positive. In total the parsing library contains about 4400 lines of Haskell. The specification of the CSP-M syntax is clearly separated from

the generation of the results (for Prolog/Java/Haskell). The performance of our CSP-M parser is definitely sufficient for our applications. For the specification of the alternating bit protocol (`abp.csp`), which is about 12KB long, the hole process from source file to Java AST takes about 50ms on a 1.3GHz Pentium laptop. Our parser is also capable of parsing big auto generated files. For example, we used the parser to process a 3.5MB file which is the CSP transliteration of a large labelled transitions system (as described in Section 5). Just parsing of this file takes about 22 seconds. A complete run of the command-line-interface version of the parser which includes, parsing, renaming/bound variable analysis, translation to a 19.7MB Prolog file, translation to a 9.4MB Haskell file and dumping a lot of information to the terminal, takes about 1 minute and 40 seconds.

In retrospective, the syntax (and semantics) of CSP-M contains many awkward and possibly doubtful features. For example, in CSP-M '`<`' and '`>`' are used for arithmetic comparison *as well as* the beginning and end of sequences and a parser has to deal with special cases like '`<true, 2>1, false>`', i.e. a comparison inside a list of Boolean expressions. These features are also the reason why it is very difficult to rigourously describe the syntax of CSP-M in terms of, i.e. a context free grammar. Luckily, with the Parsec combinator parser library, it is not important that the language one wants parse is in $LL(k)$ or $LALR(k)$. Infinite lookahead is no problem and the language does not even have to be context-free. This flexibility turned out to be invaluable for our implementation.

5 Ensuring Compliance Via Refinement Checking

To ensure that our tool is compliant with FDR we have added over 85 CSP-M regression tests. These are a mixture of existing examples (mostly from the books [24] and [27]) and artificially constructed, contrived specifications. For every regression test, a test sequence has been stored and is checked.

Furthermore, unless the CSP example cannot be treated by FDR (e.g., because of the use of an unbounded channel type), we perform a complete two way failures-divergence refinement test using FDR. For this, we have implemented a feature in ProB to dump the computed statespace into a CSP-M file. For example, the statespace on the left of Figure 2 is converted into the following:[5]

```
include "ComplicatedSync.csp"
assert MAIN [FD= Nroot
assert Nroot [FD= MAIN
Nroot =   (  {- start_cspm_MAIN-> -} N8 )
N8 =  ( ch.1.2->N9 )
N9 =  ( out2.2->N10 )
N10 =  ( ch.2.1->N11 )
N11 =  ( out.1->N12 )
N12 =  ( ch.1.2->N9 )
```

[5] `start_cspm_MAIN` is an artificial event of the ProB animator. Our translator automatically comments those events out.

By running FDR in batch mode on this file, we perform a two way failures-divergence refinement check between the original CSP-M specification (MAIN) and the statespace computed by ProB (Nroot).

More precisely, if a state i in the state space has n outgoing transitions labelled with $e_j \neq \tau$ leading to s_j and m outgoing transitions labelled with τ leading to t_k, the process definition for i will look like this, where $\triangleright$ is the "time-out" operator:[6]

$$N_i = (e_1 \rightarrow N_{s_1} \;\Box\; \ldots \;\Box\; e_n \rightarrow N_{s_n}) \;\triangleright\; (N_{t_1} \;\sqcap\; \ldots \;\sqcap\; N_{t_m})$$

Furthermore, if any of the e_j are the tick event then instead of generating $e_j \rightarrow N_{s_j}$ we generate $SKIP$ (note that N_{s_j} must be a deadlocking state) and if any of the e_j are the interrupt event `i` then we generate $STOP \triangle N_{s_j}$.

Note that this testing was very useful in uncovering bugs as well as subtle semantic misunderstandings on our behalf, and ensures a high-level of conformance of our tool. It also caught the fact that FDR does not fully comply with Roscoe's operational semantics (as far as tick and Omega are concerned). Indeed, for the following example the two-way refinement check uncovered that FDR behaved differently from our tool (and from PROBE):

```
MAIN = a->SKIP /\ b->STOP
```

Our tool computes the statespace as displayed in the right of Fig. 2. The encoding of the statespace in CSP-M is as follows:

```
include "TickOmegaFDRBug.csp"
Nroot =   (  {- start_cspm_MAIN-> -} N3 )
N3 =  ( a->N4 [] (STOP /\ N5) )
N4 =  ( SKIP  [] (STOP /\ N5) )
N5 =  ( b->N7 )
N7 = STOP
```

The Nroot process does not allow the trace $\langle a, tick, b\rangle$ which FDR computes for MAIN. The Roscoe semantics contains the following rule for the interrupt $\triangle$:

$$\frac{P \overset{tick}{\rightarrow} P'}{P \triangle Q \overset{tick}{\rightarrow} \Omega}$$

Hence, the sequence $\langle a, tick, b\rangle$ should not be allowed. After feedback from Michael Goldsmith, it turns out that FDR implements an earlier semantics (it uses $P \triangle Q \overset{tick}{\rightarrow} P' \triangle Q$), and actually behaves differently from PROBE in that respect (unless the `'-no-omega'` flag is given to PROBE on startup).

[6] Thanks to Michael Goldsmith for suggesting the use of $\triangleright$ to convert arbitrary ProB statespaces into CSP-M.

6 Empirical Evaluation

As our tool has not yet been tuned for speed, a full-scale empirical evaluation would be premature. Still, to give an idea about the strengths of our tool, we believe the following comparison with FDR to be useful. However, it should not be seen as an extensive benchmarking comparison between the two tools.

The experiments were run on MacMini with 1.83 GHz Core Duo processor running Ubuntu 7.04 with 512 MB inside of the Parallels Desktop virtualization[7] environment. All experiments involved deadlock checking of the MAIN process, and both ProB 1.2.7 and FDR 2.82 were used with default settings. Note that FDR only displays entire seconds and 0 s thus means < 0.5 s. Also, all of the non-zero timings of FDR were obtained using a stopwatch (as often the time indicated by FDR was substantially below the real time needed, as part of the precompilation time was not taken into account). The entry "**" means that FDR stopped with the message: "readEventMap failed, failed to compile ISM".

The alternating bit protocol is a sample file written by Roscoe, and accompanying Chapter 5 of [24]. Another example we tested is Crossing, a 374 line specification of a level-crossing gate, from the book webpage of [27][8]. Bank v4 is a controller accompanying a B machine from [30]. Bank Secure OZ is the model of a bank with a security automaton, translated from CSP-OZ to CSP, from [2]. This model by Bill Roscoe describes a level crossing gate using discrete-time modelling in untimed CSP. Peterson v1 and v2 are two versions from [27] of the Peterson mutual exclusion algorithm. FibGen is the example from Section 3, but with bounded channel types. I.e., this is an example of a large process composed with a small process (Take). GenPrime is an example of two processes generating numbers, and a deadlock is found if they generate the same number; the state and transition numbers are at the ones at the time when ProB found the deadlock. McCarthy0 is an example with pure recursive computation, computing the McCarthy function for $x = 0..200$ and then deadlocking. McCarthy1 and McCarthy2 are the same as McCarthy, but for $x = 0..10000$ and $x = 0..30000$. Scheduler1_CSP is the first refinement of a process scheduling specification from [17] in natural CSP style for 5 processes. Scheduler0_6_Bexp is the CSP representation of the statespace of the abstract B model of this process scheduling specification, for 6 processes. This experiment checks to what extent the tools can deal with large, simple processes (2191 CSP processes with 14601 prefix operations and 12393 external choices). Server is the example from Section 3. McCarthy1, FibGen and GenPrime can be found in Appendix A.

One can see that if a file is well tuned for FDR, then FDR can apply its compression techniques [25] very effectively, and FDR is considerably faster than ProB (e.g., Crossing). On the other hand, for specifications which are not specifically designed for FDR, the speed difference can be as dramatic, but in the favour of our tool. E.g., our tool deals better when subprocesses are large, but the whole system is not (FibGen). Our tool also seems to be faster at dealing with

[7] This was to be fair wrt FDR, as FDR 2.82 was at the time only available as a PPC version for Mac.

[8] `http://www.cs.rhul.ac.uk/books/concurrency/`

Table 1. Empirical Results for Deadlock Checking

Specification	States	Transitions	ProB	FDR
Alternating bit protocol	1852	4415	2.17 s	0 s
Bank v4	892	1231	0.53 s	0 s
Bank Secure OZ	322	3556	18.84 s	5 s
Crossing	5517	12737	53.93 s	0 s
FibGen	8	7	0.00 s	132 s
GenericBuffer1	24	28	0.01 s	12 s
GenPrime	1240	2258	0.75 s	142 s
McCarthy0	202	201	0.40 s	4 s
McCarthy1	10002	10001	4.04 s	17 s
McCarthy2	30002	30001	14.96 s	**
Peterson v1	58	115	0.40 s	0 s
Peterson v2	215	429	1.23 s	0 s
Scheduler1_CSP	4174	18031	8.66 s	2 s
Scheduler0_6_Bexp	2189	14581	2.19 s	6 s
Server	14	17	0.01 s	**

large automatically generated CSP files (Scheduler0_6_Bexp) and possibly with recursion (McCarthy0). FDR seems to have a problem with larger state spaces (McCarthy1 and McCarthy2; computing the McCarthy function for x>100 involves no recursion).

We can conclude that there are extreme differences between the two tools, and that they thus complement each other quite nicely.

7 More Related Work, Discussion and Conclusion

Other interesting tools for CSP-M are Casper [21] and CSP-Prover [12]. Apart from our own earlier work [15,5], our combined CSP and B tool is most strongly related to the csp2B tool [3]. The csp2B tool allows specifications to be written in a combination of CSP and B by compiling the CSP to a pure B representation which can be analysed by a standard B tool. The CSP supported by csp2B covers a small subset of CSP. On the Z and CSP side there is the model checker for Circus [31]. As far as using Prolog for process algebras is concerned, the model checking system XMC implemented in XSB Prolog contains an interpreter for value-passing CCS [23]. This version of CCS is much smaller in scope than CSP-M.

We have covered almost all of the CSP-M language in our interpreter, but a few constructs have not yet been implemented at the time of writing. Basically, only two primitives (extensions and productions) are not yet supported, some restrictions on channel input patterns apply[9] and mixing of closure operations with set operations (especially diff) is not yet fully supported. On the other hand, we do support some valid CSP syntax which generate errors with

[9] They can only contain variables, tuples, integers and constants; e.g., the doubtful `ch?(y+1,x)` is not accepted by our tool, but is accepted by FDR.

FDR.[10] Also, we have not yet tuned our tool for speed. But as the empirical results have shown, the tool is already applicable to realistic specifications. In future, we also plan to apply some of ProB' symmetry reduction techniques [18,19] to CSP-M.

In conclusion, we have presented a new FDR-compliant tool for validating CSP-M specifications, with many unique features making it complementary to existing tools. It is much broader in scope than the implementation in [15,5]. While developing the tool, we have uncovered some problematic issues with CSP-M, such as multiple meanings of the dot operator and that substitutivity of equals does not hold. We foresee the following potential practical applications of our new tool:

- Validation of combined B and CSP specifications. To our knowledge our toolset is unique with respect to this capability. In future it should even be possible to validate combined Z and CSP specifications as well as combined Event-B and CSP specifications within the the Rodin [4] platform [1]. Note that one can also use ProB's refinement checker to check pure B specifications against pure CSP specifications for property validation; see [5].
- Teaching process algebras and CSP. Source code highlighting of events is especially useful for beginners. ProB's interface and graphical visualisation have also proven to be very useful in a teaching environment.
- Existing CSP applications, as a complement to FDR, so as to find certain errors more quickly and highlighting errors directly in the source code, and for the first time providing an LTL model checker [20] for CSP.
- Supporting new kinds of specifications, employing infinite state processes or sub-processes.
- Safety critical applications where the specification needs to be validated by independently developed tools.

Acknowledgements

We are grateful for feedback on our new tool from Neil Evans, Steven Schneider, Helen Treharne, Edd Turner and for feedback on our paper from Daniel Plagge and anonymous referees. We are also grateful to Phil Armstrong, Michael Goldsmith, and Bill Roscoe, for insights about the semantics of CSP-M.

References

1. Abrial, J.-R., Butler, M., Hallerstede, S.: An open extensible tool environment for Event-B. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 588–605. Springer, Heidelberg (2006)

[10] E.g., FDR sometimes generates an error "unsupported comparison processing communication error" when using an if statement on parts of associative tuples received on a channel.

2. Basin, D.A., Olderog, E.-R., Sevinç, P.E.: Specifying and analyzing security automata using csp-oz. In: Bao, F., Miller, S. (eds.) ASIACCS, pp. 70–81. ACM, New York (2007)
3. Butler, M.: csp2B: A practical approach to combining CSP and B. Formal Aspects of Computing 12, 182–198 (2000)
4. Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.): Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157. Springer, Heidelberg (2006)
5. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
6. Fontaine, M., Leuschel, M.: Typechecking csp specifications using haskell (extended abstract). In: Proceedings Avocs 2007, Oxford, UK, pp. 171–176 (2007)
7. Formal Systems (Europe) Ltd. Failures-Divergence Refinement — FDR2 User Manual (version 2.8.2)
8. Formal Systems (Europe) Ltd. Process Behaviour Explorer (ProBE User Manual, version 1.30), http://www.fsel.com/probe_manual.html
9. Hill, P., Gallagher, J.: Meta-programming in logic programming. In: Gabbay, D.M., Hogger, C.J., Robinson, J.A. (eds.) Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 421–497. Oxford Science Publications, Oxford University Press, Oxford (1998)
10. Hoare, C.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
11. Hutton, G., Meijer, E.: Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham (1996)
12. Isobe, Y., Roggenbach, M.: A generic theorem prover of CSP refinement. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 108–123. Springer, Heidelberg (2005)
13. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Jouannaud, J.-P. (ed.) FPCA 1985. LNCS, vol. 201. Springer, Heidelberg (1985)
14. Jones, S.P.: The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs (1987)
15. Leuschel, M.: Design and implementation of the high-level specification language CSP(LP) in Prolog. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 14–28. Springer, Heidelberg (2001)
16. 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)
17. Leuschel, M., Butler, M.: Automatic refinement checking for B. In: Lau, K.-K., Banach, R. (eds.) ICFEM 2005. LNCS, vol. 3785, pp. 345–359. Springer, Heidelberg (2005)
18. Leuschel, M., Butler, M., Spermann, C., Turner, E.: Symmetry reduction for B by permutation flooding. In: Julliand, J., Kouchnarenko, O. (eds.) B 2007. LNCS, vol. 4355, pp. 79–93. Springer, Heidelberg (2006)
19. Leuschel, M., Massart, T.: Efficient approximate verification of B via symmetry markers. In: Proceedings International Symmetry Conference, Edinburgh, UK, January 2007, pp. 71–85 (2007)
20. Leuschel, M., Plagge, D.: Seven at a stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. In: Ameur, Y.A., Boniol, F., Wiels, V. (eds.) Proceedings Isola 2007. Revue des Nouvelles Technologies de l'Information, vol. RNTI-SM-1, Cépaduès-Éditions (2007)

21. Lowe, G.: Casper: A compiler for the analysis of security protocols. Journal of Computer Security 6(1-2), 53–84 (1998)
22. Naish, L.: An introduction to MU-Prolog. Technical Report 82/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, March 1982 (Revised, July 1983)
23. Ramakrishna, Y.S., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A., Swift, T., Warren, D.S.: Efficient model checking using tabled resolution. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 143–154. Springer, Heidelberg (1997)
24. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1999)
25. 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: TACAS 2005, pp. 133–152 (1995)
26. Scattergood, J.B.: Tools for CSP and Timed-CSP. PhD thesis, Oxford University (1997)
27. Schneider, S.: Concurrent and Real-time Systems: The CSP Approach. Wiley, Chichester (1999)
28. Steria, F.: Aix-en-Provence. Atelier B, User and Reference Manuals (1996), http://www.atelierb.societe.com
29. Treharne, H., Schneider, S.: How to drive a B machine. In: Bowen, J.P., Dunne, S., Galloway, A., King, S. (eds.) B 2000, ZUM 2000, and ZB 2000. LNCS, vol. 1878, pp. 188–208. Springer, Heidelberg (2000)
30. Treharne, H., Schneider, S., Bramble, M.: Composing specifications using communication. In: Bert, D., Bowen, J.P., King, S., Waldén, M.A. (eds.) ZB 2003. LNCS, vol. 2651, pp. 58–78. Springer, Heidelberg (2003)
31. Woodcock, J., Cavalcanti, A., Freitas, L.: Operational semantics for model checking circus. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 237–252. Springer, Heidelberg (2005)

A Some Benchmark Files

McCarthy1

```
channel out:{0..999999}
McCarthy(n) = if n>100 then
                  n-10
              else
                 McCarthy(McCarthy(n+11))
Test(n,m) = if n<m then out!McCarthy(n) -> Test(n+1,m) else STOP
MAIN = Test(0,10000)
```

FibGen

```
channel out,gen: {0..9999}
FibGen(N,M) = if M<10000 then out!M -> FibGen(M,N+M) else STOP
Take(n) = if n>1 then out?_ -> Take(n-1)
                 else out?x -> gen!x -> STOP
MAIN = FibGen(0,1) [| {| out |} |] Take(5)
```

GenPrime

```
channel out,comm:{0..99999}
MAIN =  Gen(99999,7) [| {| comm |} |] Gen(99998,29)
Gen(x,d) = out!x -> if x<d then Gen(x,d) else Gen(x-d,d)
          []
          comm.x -> STOP
```

Practical Automated Partial Verification of Multi-paradigm Real-Time Models*

Carlo A. Furia[1], Matteo Pradella[2], and Matteo Rossi[1]

[1] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
[2] CNR IEIIT-MI, Milano, Italy
{furia,pradella,rossi}@elet.polimi.it
http://home.dei.polimi.it/lastname/

Abstract. This article introduces a fully automated verification technique that permits to analyze real-time systems described using a continuous notion of time and a mixture of operational (i.e., automata-based) and descriptive (i.e., logic-based) formalisms. The technique relies on the reduction, under reasonable assumptions, of the continuous-time verification problem to its discrete-time counterpart. This reconciles in a viable and effective way the dense/discrete and operational/descriptive dichotomies that are often encountered in practice when it comes to specifying and analyzing complex critical systems. The article investigates the applicability of the technique through a significant example centered on a communication protocol. Concurrent runs of the protocol are formalized by parallel instances of a Timed Automaton, while the synchronization rules between these instances are specified through Metric Temporal Logic formulas, thus creating a multi-paradigm model. Verification tests run on this model using a bounded satisfiability checker implementing the technique show consistent results and interesting performances.

Keywords: Metric temporal logic, timed automata, discretization, dense time, bounded model checking.

1 Introduction

There is a tension between the standpoints of modeling and of verification when it comes to choosing a formal notation. The ideal modeling language would be very expressive, thus capturing sophisticated features of systems in a natural and straightforward manner; in particular, for concurrent and real-time systems, a dense time model is the intuitive choice to model true asynchrony. On the other hand, expressiveness is often traded off against complexity (and decidability), hence the desire for a feasible and fully automated verification process pulls in the opposite direction of more primitive, and less expressive, models of time and systems. Discrete time, for instance, is usually more amenable to automated verification, and quite mature techniques and tools can be deployed to verify systems modeled under this assumption.

* Work partially supported by FME Small Project and by the MIUR FIRB project "Applicazioni della Teoria degli Automi all'Analisi, Compilazione e Verifica di Software Critico e in Tempo Reale".

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 298–317, 2008.

Another, orthogonal, concern of the real-time modeler is the choice between operational and descriptive modeling languages. Typical examples of operational notations are Timed Automata (TA) and Timed Petri Nets, while temporal logics are popular instances of descriptive notations. Operational and descriptive notations have complementary strengths and weaknesses. For instance, temporal logics are very effective for describing partial models or requirements about the past (through the natural use of past operators); automata-based notations, on the other hand, model systems through the notions of state and transition, and are typically easy to simulate and visualize. From a modeling viewpoint, the possibility of integrating multiple modeling paradigms in formalizing a system would be highly desirable.

This paper introduces a verification technique that, under suitable assumptions, reconciles the dense/discrete and operational/descriptive dichotomies in an effective way. Its goal is to provide a practical means to carry out verification of real-time systems described using a dense notion of time and a mixture of operational and descriptive notations. This approach both permits to analyze continuous-time models using fully automated, discrete-time verification techniques, and allows users to mix operational (TA) and descriptive (Metric Temporal Logic, MTL) formalisms in the same specification. The technique involves an automated translation of the operational component into temporal logic notation. The resulting MTL model, which describes both the system and the properties to be verified, is then discretized according to the technique introduced in [9]. The technique is partial in two respects: it can fail to provide conclusive answers, and only dense-time behaviors with bounded variability are verified. The most common approaches to similar verification problems are in fact usually complementary, and involve translating the logic into automata [2]. Our choice is mainly justified by the fact that logic formulas are composable through conjunction, which facilitates our ultimate goal of formally combining heterogeneous models.

In this article, we start by providing a dense-time MTL axiomatization of TA. Due to a well-known expressiveness gap between temporal logics and automata [13] in general it is impossible to describe the language accepted by a TA through an MTL formula. What we provide is instead an MTL formalization of the *accepting runs* of a TA; i.e., we model the overall behavior of TA through a set of MTL axioms. It is well-known that MTL is undecidable over dense time [4]; however, this obstacle can be mitigated in practice through the *discretization* technique introduced — and demonstrated to be practically appealing — in [9]. The undecidability of dense-time MTL entails that the reduction technique must be incomplete, i.e., there are cases in which we are unable to solve the verification problem in a conclusive manner. However, as shown in [9], the impact of this shortcoming can be reduced in many practical cases. We then show that this approach yields poor results if done naïvely. Hence, we carefully revise the axiomatization and put it in a form that is much more amenable to discretization, obtaining a set of discretized MTL formulas describing TA runs. These axioms can then be combined with other modules written in MTL, and with the properties to be verified. The resulting model can be analyzed by means of automated discrete-time tools; the results of this analysis are then used to finally infer results about the verification of the original dense-time model. We provide an implementation based on the ℤot bounded satisfiability checker [17].

To investigate the effectiveness of the technique, we experimented with a significant example centered on the description of a communication protocol by means of a TA. Concurrent runs of the protocol are formalized by parallel instances of the same automaton; additionally, the synchronization rules between these instances are formalized by means of MTL formulas, thus building a multi-paradigm model. Verification tests were run on these models using the ℤot-based tool. The experimental results are encouraging, both in terms of performances and in terms of "completeness coverage" of the method.

In fact, our approach aims at providing a *practical* approach to the verification of multi-paradigm models. Hence, we sacrifice completeness in order to have a lightweight and flexible technique. Also note that, although in this paper TA are the operational formalism of choice, the same approach could be applied to other operational formalisms, such as Timed Petri Nets.

The paper is organized as follows. Section 1.1 briefly summarizes some related research. Section 2 introduces the technical definitions that are needed in the remainder, namely the syntax and semantics of MTL and TA, and the discretization technique from [11,9] that will be used. Section 3 shows how to formalize the behavior of TA as a set of dense-time MTL formulas. Then, Section 4 re-examines the axioms and suitably modifies them in a way which is most amenable to the application of the discretization technique. Section 5 describes the example of a simple communication protocol and reports on the experiments conducted on it with the SAT-based implementation of the technique. Finally, Section 6 draws some conclusions.

1.1 Related Work

To the best of our knowledge, our approach is rather unique in combining operational and descriptive formalisms over dense time, trading-off verification completeness against better performance and practical verification results. On the other hand, each of the "ingredients" of our method has been studied in isolation in the literature. In this section we briefly recall a few of the most important results in this respect.

Dense-time verification of operational models is an active field, and it has produced a few high-performance tools and methods. Let us mention, for instance, Uppaal [15] for the verification of TA. Although tools such as Uppaal exploit a descriptive notation to express the properties to be verified, the temporal logic subset is quite simple and of limited expressive power. In contrast, we allow basically full MTL to be freely used in both the description of the model and in the formalization of the properties to be verified, at the price of sacrificing completeness of verification.

MTL verification is also a well-understood research topic. MTL is known to be undecidable over dense time domains [4]. A well-known solution to this limitation restricts the syntax of MTL formulas to disallow the expression of exact (i.e., punctual) time distances [2]. The resulting logic, called MITL, is fully decidable over dense time. However, the associated decision procedures are rather difficult to implement in practice and, even if significant progress has recently been made in simplifying them [16], a serviceable implementation is still lacking.

Another approach to circumvent the undecidability of dense-time MTL builds upon the fact that the same logic is decidable over discrete time. A few approaches introduce

some notion of discretization, that is partial reduction of the verification problem from dense to discrete time. The present paper goes in this direction by extending previous work on MTL [9] to the case of TA. A different discretization technique, based on the notion of robust satisfiability of MTL specifications, has been introduced in [6]. Other work also deals with notions of robustness in order to guarantee that dense-time TA are implementable with non-ideal architectures [5]. Another well-known notion of discretization is the one based on the concept of *digitization* [12], which has been applied by several authors to the practical verification of descriptive or operational formalisms. The interested reader may also see the related work section of [9] for a more thorough comparison of other discretization techniques.

2 Preliminaries and Definitions

2.1 Behaviors

Real-time system models describe the temporal behavior of some basic items and propositions, which represent the observable "facts" of the system. More precisely, an item it is characterized by a finite domain $\mathcal{D}^{\mathsf{it}}$ (and we write $\mathsf{it} : \mathcal{D}^{\mathsf{it}}$) such that at any instant of time it takes one of the values in $\mathcal{D}^{\mathsf{it}}$. On the other hand, a proposition p is simply a fact which can be true or false at any instant of time.

A *behavior* is a formal model of a *trace* (or *run*) of some real-time system. Given a time domain $\mathbb{T}$, a finite set $\mathcal{P}$ of atomic propositions, and a finite set of items $\mathcal{I}$, a behavior b is a mapping $b : \mathbb{T} \to \mathcal{D}^{\mathsf{it}_1} \times \mathcal{D}^{\mathsf{it}_2} \times \cdots \times \mathcal{D}^{\mathsf{it}_{|\mathcal{I}|}} \times 2^{\mathcal{P}}$ which associates with every time instant $t \in \mathbb{T}$ the tuple $b(t) = \langle v_1, v_2, \ldots, v_{|\mathcal{I}|}, P \rangle$ of item values and propositions that are true at t. $\mathcal{B}_{\mathbb{T}}$ denotes the set of all behaviors over $\mathbb{T}$, for an implicit fixed set of items and propositions. $b(t)|_{\mathsf{it}}$ and $b(t)|_{\mathcal{P}}$ denote the projection of the tuple $b(t)$ over the component corresponding to item it and the set of propositions in $2^{\mathcal{P}}$ respectively. Also, $t \in \mathbb{T}$ is a *transition point* for behavior b if t is a discontinuity point of the mapping b. Depending on whether $\mathbb{T}$ is a discrete, dense, or continuous set, we call a behavior over $\mathbb{T}$ discrete-, dense-, or continuous-time respectively. In this paper, we consider the natural numbers $\mathbb{N}$ as discrete-time domain and the nonnegative real numbers $\mathbb{R}_{\geq 0}$ as continuous-time (and dense-) time domain.

Non-Zeno and non-Berkeley. Over dense-time domains, it is customary to consider only physically meaningful behaviors, namely those respecting the so-called non-Zeno property. A behavior b is non-Zeno if the sequence of transition points of b has no accumulation points. For a non-Zeno behavior b, it is well-defined the notions of values to the left and to the right of any transition point $t > 0$, which we denote as $b^-(t)$ and $b^+(t)$, respectively. In this paper, we are interested in behaviors with a stronger requirement, called *non-Berkeleyness*. Informally, a behavior b is non-Berkeley for some positive constant $\delta \in \mathbb{R}_{>0}$ if, for all $t \in \mathbb{T}$, there exists a closed interval $[u, u + \delta]$ of size δ such that $t \in [u, u + \delta]$ and b is constant throughout $[u, u + \delta]$. Notice that a non-Berkeley behavior (for any δ) is non-Zeno *a fortiori*. The set of all non-Berkeley dense-time behaviors for $\delta > 0$ is denoted by $\mathcal{B}^{\delta}_{\chi} \subset \mathcal{B}_{\mathbb{R}_{\geq 0}}$. In the following we always assume behaviors to be non-Berkeley, unless explicitly stated otherwise.

Syntax and semantics. From a purely semantic point of view, one can consider the model of a (real-time) system simply as a set of behaviors [3,8] over some time domain $\mathbb{T}$ and sets of items and propositions. In practice, however, every system is specified using some suitable notation. In this paper system models are represented through a mixture of MTL formulas [14,4] and TA [1,2]. The syntax and semantics of MTL and TA are defined in the following. Given an MTL formula or a TA μ, and a behavior b, we write $b \models \mu$ to denote that b represents a system evolution which satisfies all the constraints imposed by μ. If $b \models \mu$ for some $b \in \mathcal{B}_{\mathbb{T}}$, μ is called $\mathbb{T}$-satisfiable; if $b \models \mu$ for all $b \in \mathcal{B}_{\mathbb{T}}$, μ is called $\mathbb{T}$-valid. Similarly, if $b \models \mu$ for some $b \in \mathcal{B}^{\delta}_{\chi}$, μ is called χ^{δ}-satisfiable; if $b \models \mu$ for all $b \in \mathcal{B}^{\delta}_{\chi}$, μ is called χ^{δ}-valid.

2.2 Metric Temporal Logic

Let $\mathcal{P}$ be a finite (non-empty) set of atomic propositions, $\mathcal{I}$ be a finite set of items, and $\mathcal{J}$ be the set of all (possibly unbounded) intervals of the time domain $\mathbb{T}$ with rational endpoints.We abbreviate intervals with pseudo-arithmetic expressions, such as $= d$, $< d$, $\geq d$, for $[d, d]$, $(0, d)$, and $[d, +\infty)$, respectively.

MTL syntax. The following grammar defines the syntax of MTL, where $I \in \mathcal{J}$ and β is a Boolean combination of atomic propositions or conditions over items.

$$\phi ::= \beta \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \mathsf{U}_I(\beta_1, \beta_2) \mid \mathsf{S}_I(\beta_1, \beta_2) \mid \mathsf{R}_I(\beta_1, \beta_2) \mid \mathsf{T}_I(\beta_1, \beta_2)$$

In order to ease the presentation of the discretization techniques in Section 2.4, MTL formulas are introduced in a *flat* normal form where negations are pushed down to (Boolean combinations of) atomic propositions, and temporal operators are not nested. It should be clear, however, that any MTL formula can be put into this form, possibly by introducing auxiliary propositional letters [7]. The basic temporal operators of MTL are the *bounded until* U_I (and its past counterpart *bounded since* S_I), as well as its dual *bounded release* R_I (and its past counterpart *bounded trigger* T_I). The subscripts I denote the interval of time over which every operator predicates. Throughout the paper we omit the explicit treatment of past operators (i.e., S_I and T_I) as it can be trivially derived from that of the corresponding future operators. In the following we assume a number of standard abbreviations, such as $\bot$, $\top$, $\Rightarrow$, $\Leftrightarrow$, and, when $I = (0, \infty)$, we drop the subscript interval of operators.

MTL semantics. MTL semantics is defined over behaviors, parametrically with respect to the choice of the time domain $\mathbb{T}$. In particular, the definition of the basic temporal operators is the following:

$$\begin{array}{lll}
b(t) \models_{\mathbb{T}} \mathsf{U}_I(\beta_1, \beta_2) & \text{iff} & \text{there exists } d \in I \text{ such that: } b(t+d) \models_{\mathbb{T}} \beta_2 \\
 & & \text{and, for all } u \in [0, d] \text{ it is } b(t+u) \models_{\mathbb{T}} \beta_1 \\
b(t) \models_{\mathbb{T}} \mathsf{R}_I(\beta_1, \beta_2) & \text{iff} & \text{for all } d \in I \text{ it is: } b(t+d) \models_{\mathbb{T}} \beta_2 \text{ or there exists} \\
 & & \text{a } u \in [0, d) \text{ such that } b(t+u) \models_{\mathbb{T}} \beta_1 \\
b \models_{\mathbb{T}} \phi & \text{iff} & \text{for all } t \in \mathbb{T}\text{: } b(t) \models_{\mathbb{T}} \phi
\end{array}$$

We remark that a global satisfiability semantics is assumed, i.e., the satisfiability of formulas is implicitly evaluated over *all* time instants in the time domain. This permits

the direct and natural expression of most common real-time specifications (e.g., time-bounded response) without resorting to nesting of temporal operators. In addition, every generic MTL formulas with nesting temporal operators can be "flattened" to the form we introduced beforehand by introducing auxiliary propositions; in other words flat MTL and full MTL are equi-satisfiable .

Granularity. For an MTL formula ϕ, let $\mathcal{J}_\phi$ be the set of all non-null, finite interval bounds appearing in ϕ. Then, $\mathcal{D}_\phi$ is the set of positive values δ such that any interval bound in $\mathcal{J}_\phi$ is an integer if divided by δ.

Derived Temporal Operators. It is useful to introduce a number of derived temporal operators, to be used as shorthands in writing specification formulas. Those used in this paper are listed in Table 1 ($\delta \in \mathbb{R}_{>0}$ is a parameter used in the discretization techniques, discussed shortly).

We describe informally the meaning of such derived operators, focusing on future ones (the meaning of the corresponding past operators is easily derivable). $\Diamond_I(\beta)$ means that β happens within time interval I in the future. $\Box_I(\beta)$ means that β holds throughout the whole interval I in the future. $\widetilde{\bigcirc}(\beta)$ denotes that β holds throughout some non-empty interval in the strict future; in other words, if t is the current instant, there exists some $t' > t$ such that β holds over (t, t'). Similarly, $\bigcirc(\beta)$ denotes that β holds throughout some non-empty interval which includes the current instant, i.e., over some $[t, t')$. Then, $\triangle(\beta_1, \beta_2)$ describes a switch from condition β_1 to condition β_2, without specifying which value holds at the current instant. On the other hand, $\blacktriangle(\beta_1, \beta_2)$ describes a switch from condition β_1 to condition β_2 such that β_1 holds at the current instant; more precisely if $\triangle(\beta_1, \beta_2)$ holds at some instant t, $\blacktriangle(\beta_1, \beta_2)$ holds over $(t - \delta, t)$. In addition, for an item it we introduce the shorthand $\triangle(\mathsf{it}, v^-, v^+)$ for $\triangle(\mathsf{it} = v^-, \mathsf{it} = v^+)$. A similar abbreviation is assumed for $\blacktriangle(\mathsf{it}, v^-, v^+)$. Finally, we use $\mathrm{Alw}(\phi)$ to denote

Table 1. MTL derived temporal operators

OPERATOR	$\equiv$	DEFINITION
$\Diamond_I(\beta)$	$\equiv$	$\mathsf{U}_I(\top, \beta)$
$\overleftarrow{\Diamond}_I(\beta)$	$\equiv$	$\mathsf{S}_I(\top, \beta)$
$\Box_I(\beta)$	$\equiv$	$\mathsf{R}_I(\bot, \beta)$
$\overleftarrow{\Box}_I(\beta)$	$\equiv$	$\mathsf{T}_I(\bot, \beta)$
$\widetilde{\bigcirc}(\beta)$	$\equiv$	$\mathsf{U}_{(0,+\infty)}(\beta, \top) \vee (\neg\beta \wedge \mathsf{R}_{(0,+\infty)}(\beta, \bot))$
$\overleftarrow{\widetilde{\bigcirc}}(\beta)$	$\equiv$	$\mathsf{S}_{(0,+\infty)}(\beta, \top) \vee (\neg\beta \wedge \mathsf{T}_{(0,+\infty)}(\beta, \bot))$
$\bigcirc(\beta)$	$\equiv$	$\beta \wedge \widetilde{\bigcirc}(\beta)$
$\overleftarrow{\bigcirc}(\beta)$	$\equiv$	$\beta \wedge \overleftarrow{\widetilde{\bigcirc}}(\beta)$
$\triangle(\beta_1, \beta_2)$	$\equiv$	$\begin{cases} \overleftarrow{\widetilde{\bigcirc}}(\beta_1) \wedge \left(\beta_2 \vee \widetilde{\bigcirc}(\beta_2)\right) & \text{if } \mathbb{T} = \mathbb{R}_{\geq 0} \\ \overleftarrow{\Diamond}_{=1}(\beta_1) \wedge \Diamond_{[0,1]}(\beta_2) & \text{if } \mathbb{T} = \mathbb{N} \end{cases}$
$\blacktriangle(\beta_1, \beta_2)$	$\equiv$	$\begin{cases} \beta_1 \wedge \Diamond_{=\delta}(\beta_2) & \text{if } \mathbb{T} = \mathbb{R}_{\geq 0} \\ \beta_1 \wedge \Diamond_{=1}(\beta_2) & \text{if } \mathbb{T} = \mathbb{N} \end{cases}$

$\phi \wedge \Box_{(0,+\infty)}(\phi) \wedge \overleftarrow{\Box}_{(0,+\infty)}(\phi)$. Since $b \models_{\mathbb{T}} \mathrm{Alw}(\phi)$ iff $b \models_{\mathbb{T}} \phi$, for any behavior b, $\mathrm{Alw}(\phi)$ can be expressed without nesting if ϕ is flat, through the global satisfiability semantics introduced beforehand.

2.3 Operational Model: Timed Automata

We introduce a variant of TA which differs from the classical definitions (e.g., [1]) in that it recognizes behaviors, rather than timed words [2,16]. Correspondingly, input symbols are associated with locations rather than with transitions. Also, we introduce the following simplifications that are known to be without loss of generality: we do not define location clock invariants (also called staying conditions) and use transition guards only, and we forbid self-loop transitions.

We introduce one additional variant which does impact expressiveness, namely clock constraints do not distinguish between different transition edges, that is between transitions occurring right- and left-continuously. This restriction is motivated by our ultimate goal of *discretizing* TA: as it will be explained later, such distinctions would inevitably be lost in the discretization process, hence we give them up already.

Finally, for the sake of simplicity, we do not consider acceptance conditions, that is let us assume that all states are accepting. Notice that introducing acceptance conditions (e.g., Büchi, Muller, etc.) in the formalization would be routine.

TA syntax. For a set C of clock variables, the set $\Phi(C)$ of *clock constraints* ξ is defined inductively by

$$\xi ::= c < k \mid c \geq k \mid \xi_1 \wedge \xi_2 \mid \xi_1 \vee \xi_2$$

where c is a clock in C and k is a constant in $\mathbb{Q}_{\geq 0}$.

A *timed automaton* A is a tuple $\langle \Sigma, S, S_0, \alpha, C, E \rangle$, where:

- Σ is a finite (input) alphabet,
- S is a finite set of locations,
- $S_0 \subseteq S$ is a finite set of initial locations,
- $\alpha : S \to 2^{\Sigma}$ is a location labeling function that assigns to each location $s \in S$ a set $\alpha(s)$ of propositions,
- C is a finite set of clocks, and
- $E \subseteq S \times S \times 2^C \times \Phi(C)$ is a set of transitions. An edge $\langle s, s', \Lambda, \xi \rangle$ represents a transition from state s to state $s' \neq s$; the set $\Lambda \subseteq C$ identifies the clocks to be reset with this transition, and ξ is a clock constraint over C.

TA semantics. In defining the semantics of TA over behaviors we deviate from the standard presentation (e.g., [2,16]) in that we do not represent TA as acceptors of behaviors over the input alphabet Σ, but rather as acceptors of behaviors representing what are usually called *runs* of the automaton. In other words, we introduce automata as acceptors of behaviors over the items st and in representing, respectively, the current location and the current input symbol, as well as propositions $\{\mathsf{rs}_c \mid c \in C\}$ representing the clock reset status. This departure from more traditional presentations is justified by the fact that we intend to provide an MTL axiomatic description of TA runs — rather than accepted languages, which would be impossible for a well-known expressiveness gap

[13] — hence we define the semantics of automata over this "extended" state from the beginning.

Here we sketch an informal description of the semantics. Initially, all clocks are reset and the automaton is in state $s_0 \in S_0$. At any given time t, when the automaton is in some state s, it can take nondeterministically a transition $\langle s, s', \Lambda, \xi \rangle$ to some other state s', only if the last time (before t) each clock has been reset is compatible with constraint ξ. If the transition is taken, all clocks in Λ are reset, whereas all the other clocks keep on running. Finally, as long as the automaton is in any state s, the input has to satisfy the location labeling function $\alpha(s)$, namely the current input corresponds to exactly one of the propositions in $\alpha(s)$.

A timed automaton $A = \langle \Sigma, S, S_0, \alpha, C, E \rangle$ is interpreted over behaviors over items $\mathsf{st} : S, \mathsf{in} : \Sigma$ and propositions $R = \{\mathsf{rs}_c \mid c \in C\}$. At any instant of time t, $\mathsf{st} = s$ means that the automaton is in state s, $\mathsf{in} = \sigma$ means that the input symbol is σ, and rs_c keeps track of resets of clock c (we model such resets through switches, from false to true or *vice versa*, of rs_c). Let b be such a behavior, and let t be one of its transition points. Satisfaction of clock constraints at t is defined as follows:

$$
\begin{array}{lll}
b(t) \models c < k & \text{iff} & \text{either } b^-(t) \models \mathsf{rs}_c \text{ and there exists } t-k < t' < t \text{ s.t. } b(t') \not\models \mathsf{rs}_c; \\
 & & \text{or } b^-(t) \not\models \mathsf{rs}_c \text{ and there exists a } t-k < t' < t \text{ s.t. } b(t') \models \mathsf{rs}_c \\
b(t) \models c \geq k & \text{iff} & \text{either } b^-(t) \models \mathsf{rs}_c \text{ and for all } t-k < t' < t : b'(t) \models \mathsf{rs}_c; \\
 & & \text{or } b^-(t) \not\models \mathsf{rs}_c \text{ and for all } t-k < t' < t : b(t') \not\models \mathsf{rs}_c
\end{array}
$$

Notice that this corresponds to looking for the previous time the proposition rs_c switched (from false to true or from true to false) and counting time since then. This requires a little "hack" in the definition of the semantics: namely, a first start reset of all clocks is issued before the "real" run begins; this is represented by time instant t_{start} in the formal semantics below.

Formally, a behavior b over $\mathsf{st} : S, \mathsf{in} : \Sigma, R$ (with $b : \mathbb{R}_{\geq 0} \to S \times \Sigma \times 2^R$) is a *run* of the automaton A, and we write $b \models_{\mathbb{R}_{\geq 0}} A$, iff:

- $b(0) = \langle s_0, \sigma, \bigcup_{c \in C} \{\mathsf{rs}_c\} \rangle$ and $\sigma \in \alpha(s_0)$ for some $s_0 \in S_0$;
- there exists a transition instant $t_{\text{start}} > 0$[1] such that: $b(t)|_{\mathsf{st}} = s_0$ and $b(t)|_R = R$ for all $0 \leq t \leq t_{\text{start}}$, $b^-(t_{\text{start}}) = \langle s_0, \sigma^-, \rho^- \rangle$ and $b^+(t_{\text{start}}) = \langle s^+, \sigma^+, \rho^+ \rangle$ with $\rho^- = R$ and $\rho^+ = \emptyset$;
- for all $t \in \mathbb{R}_{\geq 0}$: $b(t)|_{\mathsf{in}} \in \alpha(b(t)|_{\mathsf{st}})$;
- for all transition instants $t > t_{\text{start}}$ of $b|_{\mathsf{st}}$ or $b|_R$ such that $b^-(t) = \langle s^-, \sigma^-, \rho^- \rangle$ and $b^+(t) = \langle s^+, \sigma^+, \rho^+ \rangle$, it is: $\langle s^-, s^+, \Lambda, \xi \rangle \in E$, $\sigma^- \in \alpha(s^-)$, $\sigma^+ \in \alpha(s^+)$, $\rho = \bigcup_{c \in \Lambda} \{\mathsf{rs}_c\}$, $\rho^+ = \rho^- \triangle \rho = (\rho^- \setminus \rho) \cup (\rho \setminus \rho^-)$, and $b(t) \models \xi$.

2.4 Discrete-Time Approximations of Continuous-Time Specifications

This section concisely summarizes the fundamental results from [9] that are needed in the remainder of the paper, and provides some intuition about how they can be applied to the discretization problem.

[1] In the following, we will assume that $t_{\text{start}} \in (\delta, 2\delta)$ for the discretization parameter $\delta > 0$.

The technique of [9] is based on two approximation functions for MTL formulas, called under- and over-approximation. The under-approximation function $\Omega_\delta(\cdot)$ maps dense-time MTL formulas to discrete-time formulas such that the non-validity of the latter implies the non-validity of the former, over behaviors in $\mathcal{B}^\delta_\chi$. The over-approximation function $\mathrm{O}_\delta(\cdot)$ maps dense-time MTL formulas to discrete-time MTL formulas such that the validity of the latter implies the validity of the former, over behaviors in $\mathcal{B}^\delta_\chi$. We have the following fundamental verification result, which provides a justification for the TA verification technique discussed in this paper.

Proposition 1 (Approximations [9]). *For any MTL formulas ϕ_1, ϕ_2, and for any $\delta \in \mathcal{D}_{\phi_1,\phi_2}$: (1) if $\mathrm{Alw}(\Omega_\delta(\phi_1)) \Rightarrow \mathrm{Alw}(\mathrm{O}_\delta(\phi_2))$ is $\mathbb{N}$-valid, then $\mathrm{Alw}(\phi_1) \Rightarrow \mathrm{Alw}(\phi_2)$ is χ^δ-valid; and (2) if $\mathrm{Alw}(\mathrm{O}_\delta(\phi_1)) \Rightarrow \mathrm{Alw}(\Omega_\delta(\phi_2))$ is not $\mathbb{N}$-valid, then $\mathrm{Alw}(\phi_1) \Rightarrow \mathrm{Alw}(\phi_2)$ is not χ^δ-valid.*

Discussion. Proposition 1 suggests a verification technique which builds two formulas through a suitable composition of over- and under-approximations of the system description and the putative properties, and it infers the validity of the properties from the results of a discrete-time validity checking. The technique is incomplete as, in particular, when approximation (1) is not valid and approximation (2) is valid nothing can be inferred about the validity of the property in the original system over dense time.

It is important to notice that equivalent dense-time formulas can yield dramatically different — in terms of usefulness — approximated discrete-time formulas. For instance, consider dense-time MTL formula $\theta_1 = \Box_{(0,\delta)}(\mathsf{p})$ which, under the global satisfiability semantics, says that p is *always* true. Its under-approximation is $\Omega_\delta(\theta_1) = \Box_\emptyset(\mathsf{p})$ which holds for any discrete-time behavior! Thus, we have an under-approximation which is likely too coarse, as it basically adds no information to the discrete-time representation. So, if we build formula (1) from Proposition 1 with $\Omega_\delta(\theta_1)$ in it, it is likely that the antecedent will be trivially satisfiable (because $\Omega_\delta(\theta_1)$ introduces no constraint) and hence formula (1) will be non-valid, yielding no information to the verification process. If, however, we modify θ_1 into the *equivalent* $\theta_1' = \mathsf{p} \wedge \theta_1$ we get an under-approximation which can be written simply as $\Omega_\delta(\theta_1') = \mathsf{p}$, which correctly entails that p is always true over discrete-time as well. This is likely a much better approximation, one which better preserves the original "meaning" of θ_1.

3 Formalizing Timed Automata in MTL

Consider a TA $A = \langle \Sigma, S, S_0, \alpha, C, E \rangle$; this section introduces an MTL formalization of the runs of A over non-Berkeley behaviors, for some $\delta > 0$. In other words, this section provides a set of formulas $\phi_1, \ldots, \phi_6$ such that, for all non-Berkeley behaviors b, $b \models A$ iff $b \models \phi_j$ for all $j = 1, \ldots, 6$.

Clock constraints. Given a clock constraint ξ, we represent by $\Xi(\xi)$ an MTL formula such that $b(t) \models \xi$ iff $b(t) \models \Xi(\xi)$ at all transition points t. $\Xi(\xi)$ can be defined inductively as:

$$
\begin{array}{lcl}
\Xi\,(c < k) & \equiv & \overleftarrow{\bigcirc}(\mathsf{rs}_c) \wedge \overleftarrow{\Diamond}_{(0,k)}(\neg\mathsf{rs}_c) \;\vee\; \overleftarrow{\bigcirc}(\neg\mathsf{rs}_c) \wedge \overleftarrow{\Diamond}_{(0,k)}(\mathsf{rs}_c) \\
\Xi\,(c \geq k) & \equiv & \overleftarrow{\bigcirc}(\mathsf{rs}_c) \wedge \overleftarrow{\Box}_{(0,k)}(\mathsf{rs}_c) \;\vee\; \overleftarrow{\bigcirc}(\neg\mathsf{rs}_c) \wedge \overleftarrow{\Box}_{(0,k)}(\neg\mathsf{rs}_c) \\
\Xi\,(\xi_1 \wedge \xi_2) & \equiv & \Xi\,(\xi_1) \wedge \Xi\,(\xi_1) \\
\Xi\,(\xi_1 \vee \xi_2) & \equiv & \Xi\,(\xi_1) \vee \Xi\,(\xi_1)
\end{array}
$$

Essentially, Ξ translates guard ξ by comparing the current time to the last time a reset for the clock c happened, where a reset is represented by a switching of item rs_c. Notice that, to compute the approximations of the clock-constraint formulas, every constant k used in the definition of the TA must be an integral multiple of δ.

Necessary conditions for state change. Let us state the necessary conditions that characterize a state change. For any pair of states $s_i, s_j \in S$ such that there are K transitions $\langle s_i, s_j, \Lambda^k, \xi^k \rangle \in E$ for all $1 \leq k \leq K$, we introduce the following axiom:

$$
\triangle(\mathsf{st}, s_i, s_j) \quad \Rightarrow \quad \bigvee_k \left(\Xi(\xi^k) \wedge \bigwedge_{c \in \Lambda^k} \Big(\triangle(\neg\mathsf{rs}_c, \mathsf{rs}_c) \vee \triangle(\mathsf{rs}_c, \neg\mathsf{rs}_c) \Big) \right) \tag{1}
$$

Also, we introduce an axiom asserting that, for any pair of states $s_i \neq s_j \in S$ such that $\langle s_i, s_j, \Lambda, \xi \rangle \notin E$ for any Λ, ξ (i.e., for any pair of states that are not connected by any edge), there cannot be a transition from s_i to s_j:

$$
\neg\triangle(\mathsf{st}, s_i, s_j) \tag{2}
$$

Sufficient conditions for state change. There are multiple sufficient conditions for state changes; basically, they account for reactions to reading input symbols and resetting clocks. Let us consider input first: the staying condition in every state must be satisfied always, so for all $s \in S$ the following axiom is added:

$$
\mathsf{st} = s \quad \Rightarrow \quad \mathsf{in} \in \alpha(s) \tag{3}
$$

Then, for each reset of a clock $c \in C$, for all $1 \leq k \leq K$ such that $\langle s_i^k, s_j^k, \Lambda^k, \xi^k \rangle \in E$ is an edge such that $c \in \Lambda^k$ (i.e., on which c is reset), the following axiom is introduced (a similar one for the transition of rs_c from true to false is also included):

$$
\triangle(\neg\mathsf{rs}_c, \mathsf{rs}_c) \quad \Rightarrow \quad \bigvee_k \triangle\big(\mathsf{st}, s_i^k, s_j^k\big) \tag{4}
$$

Initialization and liveness condition. The axiomatization is completed by including formulas describing initialization and liveness conditions. The following axiom, describing system initialization, is only evaluated at 0:

$$
\text{at } 0\text{:} \quad \bigwedge_{c \in C} \mathsf{rs}_c \wedge \Diamond_{[0,2\delta]} \left(\bigwedge_{c \in C} \neg\mathsf{rs}_c \right) \wedge \bigvee_{s_0 \in S_0} \bigcirc(\mathsf{st} = s_0) \tag{5}
$$

Finally, we introduce a "liveness" condition stating that the automaton must eventually move out of every state. Thus, for every state $s \in S$, if $S'_s \subset S$ is the set of states that

are directly reachable from s through a single transition the following axiom asserts that, if the automaton is in s, it must eventually move to a state in S'_s:

$$\mathsf{st} = s \;\Rightarrow\; \Diamond\left(\bigvee_{s' \in S'_s} \mathsf{st} = s'\right) \tag{6}$$

Since axiom (6) does not mandate that only some particular states must be traversed infinitely often, this corresponds to the condition that all states are accepting *à la* Büchi.

The next proposition states the axiomatization correctness (see [10] for details).

Proposition 2 (MTL TA Axiomatization). *Let $A = \langle \Sigma, S, S_0, \alpha, C, E\rangle$ be a timed automaton, $\phi_1^A, \ldots, \phi_6^A$ be formulas (1–6) for TA A, and let $b \in \mathcal{B}_\chi^\delta$ be any non-Berkeley behavior over items $\mathsf{st} : S, \mathsf{in} : \Sigma$ and propositions in R. Then $b \models A$ if and only if $b \models \bigwedge_{1 \le j \le 6} \phi_j^A$.*

4 Discrete-Time Approximations of Timed Automata

While formulas (1–6) correctly formalize the behavior of TA as defined in Section 2.3, they yield approximations of little use for verification purposes, since they are very likely to produce inconclusive results due to the incompleteness of the technique. To avoid such problems, instead of computing approximations directly from axioms (1–6), we introduce new formulas, which are equivalent to (1–6) over non-Berkeley behaviors, but whose form yields better approximations. In the rest of this section we first compute the under-approximation of formulas (1–6) (Section 4.1), and then their over-approximation (Section 4.2).

4.1 Under-Approximation

As mentioned above, the form of some of the axioms (1–6) produces under-approximations that are ill-suited to perform verification through the discretization technique of [9], due to the inherent incompleteness of the latter.

While the details underlying this issue are outside the scope of this article (and can be found in [10]), let us hint at some of the problems that arise from the under-approximation of formulas (1–6). First, it can be shown that, in general, $\Omega_\delta(\neg\triangle(\beta_1, \beta_2)) \neq \neg\Omega_\delta(\triangle(\beta_1, \beta_2))$; since subformulas of the form $\triangle(\mathsf{st} = s_i, \mathsf{st} = s_j)$ are used in (1–6) to describe state transitions, there are discrete-time behaviors where such a transition both occurs and does not occur, i.e., $\Omega_\delta(\triangle(\mathsf{st} = s_i, \mathsf{st} = s_j))$ and $\Omega_\delta(\neg\triangle(\mathsf{st} = s_i, \mathsf{st} = s_j))$ are both true, which is an approximation too coarse to be useful. Second, $\Omega_\delta(\neg\triangle(\beta_1, \beta_2))$ is a very weak formula, in that it can be shown to be true, in particular, whenever β_1 or β_2 are false; then, antecedents $\triangle(\mathsf{st} = s_i, \mathsf{st} = s_j)$ in (1–6) are trivially true because it can never be that both $\mathsf{st} = s_i$ and $\mathsf{st} = s_j$ when $s_i \neq s_j$.

To obtain better approximations, in the new axiomatization every occurrence of $\triangle(\beta_1, \beta_2)$ is replaced with $\blacktriangle(\beta_1, \beta_2)$. This entails that formulas $\Xi(\xi)$ representing clock constraints must also be changed in $\overrightarrow{\Xi}(\xi)$, where $\overrightarrow{\Xi}$ is defined below. Hence, formulas (1–2),(4) become:

$$\blacktriangle(\mathsf{st}, s_i, s_j) \quad \Rightarrow \quad \bigvee_k \overrightarrow{\Xi}(\xi^k) \ \wedge \ \bigwedge_{c\in\Lambda^k} \Big(\blacktriangle(\neg\mathsf{rs}_c, \mathsf{rs}_c) \ \vee \ \blacktriangle(\mathsf{rs}_c, \neg\mathsf{rs}_c)\Big) \tag{7}$$

$$\neg\blacktriangle(\mathsf{st}, s_i, s_j) \tag{8}$$

$$\blacktriangle(\neg\mathsf{rs}_c, \mathsf{rs}_c) \quad \Rightarrow \quad \bigvee_k \blacktriangle\big(\mathsf{st}, s_i^k, s_j^k\big) \tag{9}$$

where $\overrightarrow{\Xi}$ is defined as follows:

$$\begin{aligned}
\overrightarrow{\Xi}(c < k) &\equiv \mathsf{rs}_c \wedge \overleftarrow{\Diamond}_{(0,k)}(\neg\mathsf{rs}_c) \quad \vee \quad \neg\mathsf{rs}_c \wedge \overleftarrow{\Diamond}_{(0,k)}(\mathsf{rs}_c)\\
\overrightarrow{\Xi}(c \geq k) &\equiv \mathsf{rs}_c \wedge \overleftarrow{\Box}_{(0,k-\delta)}(\mathsf{rs}_c) \quad \vee \quad \neg\mathsf{rs}_c \wedge \overleftarrow{\Box}_{(0,k-\delta)}(\neg\mathsf{rs}_c)
\end{aligned}$$

It can be shown that, given a non-Berkeley behavior $b \in \mathcal{B}_\chi^\delta$, $b \models$ (1) iff $b \models$ (7), $b \models$ (2) iff $b \models$ (8) and $b \models$ (4) iff $b \models$ (9).

Proof. Let us first show that (1) implies (7), so let t be the current instant, assume that (1) and the antecedent $\blacktriangle(\mathsf{st}, s_i, s_j)$ of (7) hold: we establish that the consequent of (7) holds. $\blacktriangle(\mathsf{st}, s_i, s_j)$ means that $\mathsf{st} = s_i$ at t and $\mathsf{st} = s_j \neq s_i$ at $t+\delta$; hence there must be a transition instant t' of item st somewhere in $[t, t+\delta]$. Then (1) evaluated at t' entails that t' is a transition instant for some propositions $\mathsf{rs}_c|_{c\in\Lambda^k}$ as well. Let $d \in C$ be anyone of such clocks and assume that $\triangle(\mathsf{rs}_d, \neg\mathsf{rs}_d)$ holds at t'. Let us first assume $t' \in (t, t+\delta)$; correspondingly, from the non-Berkeleyness assumption, rs_d holds over $[t, t')$ and $\neg\mathsf{rs}_d$ holds over $(t', t+\delta]$. In particular, rs_d holds at t and $\neg\mathsf{rs}_d$ holds at $t+\delta$, so $\blacktriangle(\mathsf{rs}_d, \neg\mathsf{rs}_d)$ holds at t. Otherwise, let $t' = t$, so st changes its value left-continuously at t. Then, again from (1) and the non-Berkeleyness assumption, rs_d also changes its value left-continuously, so rs_d holds at t and $\neg\mathsf{rs}_d$ holds at $t+\delta$. Finally, if $t' = t+\delta$, st changes its value right-continuously at t', so rs_d also changes its value right-continuously, so rs_d holds at t and $\neg\mathsf{rs}_d$ holds at $t+\delta$. In all, since d is generic, and the same reasoning applies for the converse transition $\triangle(\neg\mathsf{rs}_d, \mathsf{rs}_d)$, we have established that $\bigwedge_{c\in\Lambda^k}(\blacktriangle(\neg\mathsf{rs}_c, \mathsf{rs}_c) \vee \blacktriangle(\mathsf{rs}_c, \neg\mathsf{rs}_c))$ holds at t.

Next, let us establish $\overrightarrow{\Xi}(\xi^k)$ from $\Xi(\xi^k)$. Let us first consider some $\Xi(d < k)$ such that $\overleftarrow{\bigcirc}(\mathsf{rs}_d) \wedge \overleftarrow{\Diamond}_{(0,k)}(\neg\mathsf{rs}_d)$ at t'. So, let $t'' \in (t'-k, t')$ be the largest instant with a transition from $\neg\mathsf{rs}_d$ to rs_d. Note that it must actually be $t'' \in (t'-k, t]$ because $t' - t \leq \delta$ and the non-Berkeleyness assumption. If $t'' \in (t'-k, t) \subseteq (t-k, t)$ then $\mathsf{rs}_d \wedge \overleftarrow{\Diamond}_{(0,k)}(\neg\mathsf{rs}_d)$ holds at t, hence $\overrightarrow{\Xi}(d < k)$ is established. If $t'' = t$ then rs_d switches to true right-continuously at t, so $\mathsf{rs}_d \wedge \overleftarrow{\bigcirc}(\neg\mathsf{rs}_d)$ at t which also entails $\overrightarrow{\Xi}(d < k)$. The same reasoning applies if $\overleftarrow{\bigcirc}(\neg\mathsf{rs}_c) \wedge \overleftarrow{\Diamond}_{(0,k)}(\mathsf{rs}_c)$ holds at t'. Finally, consider some $\Xi(d \geq k)$ such that $\overleftarrow{\bigcirc}(\mathsf{rs}_d) \wedge \overleftarrow{\Box}_{(0,k)}(\mathsf{rs}_d)$ holds at t, thus rs_d holds over $(t-k, t)$. From $t \leq t' + \delta$ we have $t' + \delta - k \geq t + k$ so $(t' - k + \delta, t') \subseteq (t-k, t)$, which shows that $\overleftarrow{\Box}_{(0,k-\delta)}(\mathsf{rs}_d)$ holds at t'. The usual reasoning about transition edges would allow us to establish that also rs_d holds at t'. Since the same reasoning applies

if $\widetilde{\overleftarrow{\bigcirc}}(\neg \mathsf{rs}_d) \wedge \overleftarrow{\Box}_{(0,k)}(\neg \mathsf{rs}_d)$, we have established that $\overrightarrow{\Xi}(d \geq k)$ holds at t'. Since d is generic, we have that $\overrightarrow{\Xi}(\xi^k)$ holds at t'.

Let us now prove (7) implies (1), so let t be the current instant, assume that (7) and the antecedent $\triangle(\mathsf{st}, s_i, s_j)$ of (1) hold: we establish that the consequent of (1) holds. So, there is a transition of st from s_i to $s_j \neq s_i$ at t; from the non-Berkeleyness assumption we have that $\mathsf{st} = s_i$ and $\mathsf{st} = s_j$ hold over $[t-\delta, t)$ and $(t, t+\delta]$, respectively. If the transition of st is left-continuous (i.e., $\mathsf{st} = s_i$ holds at t), consider (7) at t, where the antecedent holds. So, $\overrightarrow{\Xi}(\xi^k) \wedge \bigwedge_{c \in \Lambda^k} (\blacktriangle(\neg \mathsf{rs}_c, \mathsf{rs}_c) \vee \blacktriangle(\mathsf{rs}_c, \neg \mathsf{rs}_c))$ holds at t for some k. Let $d \in \Lambda^k$ be such that $\blacktriangle(\neg \mathsf{rs}_d, \mathsf{rs}_d)$ holds, that is $\neg \mathsf{rs}_d$ holds at t and rs_d holds at $t+\delta$. This entails that there exists a transition point $t' \in [t, t+\delta]$ of rs_d. However, t is already a transition point, thus it must be $t' = t$; this shows $\triangle(\neg \mathsf{rs}_d, \mathsf{rs}_d)$ at d. Recall that d is generic, and the same reasoning applies for the converse transition from rs_d to $\neg \mathsf{rs}_d$. If, instead, the transition of st is right-continuous (i.e., $\mathsf{st} = s_j$ holds at t), we consider (7) at $t - \delta$ and perform a similar reasoning. All in all, we have established that $\bigwedge_{c \in \Lambda^k} (\triangle(\neg \mathsf{rs}_c, \mathsf{rs}_c) \vee \triangle(\mathsf{rs}_c, \neg \mathsf{rs}_c))$ holds at t.

The clock constraint formula $\Xi(\xi^k)$ can also be proved along the same lines. For instance, assume that the transition of st at t is left-continuous and $\overleftarrow{\bigcirc}(\mathsf{rs}_d)$ holds at t for some $d \in C$, and consider a constraint $\overrightarrow{\Xi}(d < k)$ at t. We have that $\overleftarrow{\Diamond}_{(0,k)}(\neg \mathsf{rs}_d)$ must holds at t, which establishes that $\Xi(d < k)$ holds at t. Similar reasonings apply to the other cases. See [10] for proofs of the other equivalences. □

Then, the axiomatization of TA given by formulas (7–9),(3),(5–6) yields the following under-approximations.

$$\Omega_\delta\,((7)) \quad \equiv \quad \blacktriangle(\mathsf{st}, s_i, s_j) \quad \Rightarrow \quad \bigvee_k \Omega_\delta \left(\overrightarrow{\Xi}(\xi^k) \right) \wedge \bigwedge_{c \in \Lambda^k} \begin{pmatrix} \blacktriangle(\neg \mathsf{rs}_c, \mathsf{rs}_c) \\ \vee \\ \blacktriangle(\mathsf{rs}_c, \neg \mathsf{rs}_c) \end{pmatrix} \tag{10}$$

where:

$$\begin{aligned} \Omega_\delta \left(\overrightarrow{\Xi}\,(c < k) \right) \quad &\equiv \quad \mathsf{rs}_c \wedge \overleftarrow{\Diamond}_{[0,k/\delta]}(\neg \mathsf{rs}_c) \quad \vee \quad \neg \mathsf{rs}_c \wedge \overleftarrow{\Diamond}_{[0,k/\delta]}(\mathsf{rs}_c) \\ \Omega_\delta \left(\overrightarrow{\Xi}\,(c \geq k) \right) \quad &\equiv \quad \mathsf{rs}_c \wedge \overleftarrow{\Box}_{[1,k/\delta-2]}(\mathsf{rs}_c) \quad \vee \quad \neg \mathsf{rs}_c \wedge \overleftarrow{\Box}_{[0,k/\delta-2]}(\neg \mathsf{rs}_c) \end{aligned}$$

In addition the following can be proved to hold:

$$\Omega_\delta\,((8)) \quad \equiv \quad \neg \blacktriangle(\mathsf{st}, s_i, s_j) \tag{11}$$

$$\Omega_\delta\,((9)) \quad \equiv \quad \blacktriangle(\neg \mathsf{rs}_c, \mathsf{rs}_c) \Rightarrow \bigvee_k \blacktriangle\left(\mathsf{st}, s_i^k, s_j^k\right) \tag{12}$$

$$\Omega_\delta\,((3)) \quad \equiv \quad (3) \tag{13}$$

$$\text{at } 0\text{:} \quad \Omega_\delta\,((5)) \quad \equiv \quad \bigwedge_{c \in C} \mathsf{rs}_c \wedge \Diamond_{[1,2]} \left(\bigwedge_{c \in C} \neg \mathsf{rs}_c \right) \wedge \bigvee_{s_0 \in S_0} \mathsf{st} = s_0 \tag{14}$$

$$\Omega_\delta\,((6)) \quad \equiv \quad (6) \tag{15}$$

4.2 Over-Approximation

While the over-approximation of axioms (1–6) poses less problems than their under-approximation, it must nonetheless be carried out very carefully, and some modifications to the axioms are in order in this case, too. Notice that the following equalities hold:

- $\mathrm{O}_\delta\left(\widetilde{\bigcirc}(\beta)\right) = \Box_{[0,1]}(\beta)$.
- $\mathrm{O}_\delta\left(\Diamond_{[0,2\delta]}(\beta)\right) = \Diamond_{=1}(\beta)$.
- $\mathrm{O}_\delta\left(\bigcirc(\beta)\right) = \Box_{[0,1]}(\beta)$.
- $\mathrm{O}_\delta\left(\overleftarrow{\widetilde{\bigcirc}}(\beta)\right) = \mathrm{O}_\delta\left(\overleftarrow{\bigcirc}(\beta)\right) = \overleftarrow{\Box}_{[0,1]}(\beta)$.
- $\mathrm{O}_\delta\left(\neg\triangle(\beta_1,\beta_2)\right) = \neg(\triangle(\beta_1,\beta_2) \vee \blacktriangle(\beta_1,\beta_2))$ if $\neg(\beta_1 \wedge \beta_2)$ holds.

The over-approximations of the clock constraints (i.e., $\mathrm{O}_\delta\left(\Xi(\xi)\right)$) pose little problems when ξ is of the form $c < k$; however, when ξ is of the form $c \geq k$, they yield formulas that are unsatisfiable if there are some transitions that reset c and whose guard is $c \geq k$. Hence, the definition of $\Xi(c \geq k)$ must be modified in the following way (which is equivalent to the previous formulation for non-Berkeley behaviors):

$$\Xi\left(c \geq k\right) \quad \equiv \quad \overleftarrow{\widetilde{\bigcirc}}(\mathsf{rs}_c) \wedge \overleftarrow{\Box}_{[\delta,k)}(\mathsf{rs}_c) \;\vee\; \overleftarrow{\widetilde{\bigcirc}}(\neg\mathsf{rs}_c) \wedge \overleftarrow{\Box}_{[\delta,k)}(\neg\mathsf{rs}_c)$$

Therefore, the over-approximations of the new clock constraints are the following:

$$\mathrm{O}_\delta\left(\Xi\left(c < k\right)\right) \equiv \overleftarrow{\Box}_{[0,1]}(\mathsf{rs}_c) \wedge \overleftarrow{\Diamond}_{[1,k/\delta-1]}(\neg\mathsf{rs}_c) \vee \overleftarrow{\Box}_{[0,1]}(\neg\mathsf{rs}_c) \wedge \overleftarrow{\Diamond}_{[1,k/\delta-1]}(\mathsf{rs}_c)$$
$$\mathrm{O}_\delta\left(\Xi\left(c \geq k\right)\right) \equiv \overleftarrow{\Box}_{[0,1]}(\mathsf{rs}_c) \wedge \overleftarrow{\Box}_{[0,k/\delta+1]}(\mathsf{rs}_c) \vee \overleftarrow{\Box}_{[0,1]}(\neg\mathsf{rs}_c) \wedge \overleftarrow{\Box}_{[0,k/\delta+1]}(\neg\mathsf{rs}_c)$$

The over-approximation of formula (1), instead, is very poor verification-wise, because subformulas of the form $\triangle(\beta_1,\beta_2)$ such that β_1,β_2 cannot hold at the same instant produce over-approximations that are unsatisfiable. The same problems arise with the over-approximation of formula (4).

However, similarly to what was done in Section 4.1, it is possible to rewrite axioms (1) and (4) so that they yield better over-approximations. The two following formulas are equivalent, over non-Berkeley behaviors, to (1) and (4), respectively (see [10] for equivalence proofs).

$$\triangle(\mathsf{st}, s_i, s_j) \Rightarrow \bigvee_k \left(\Xi(\xi^k) \wedge \bigwedge\nolimits_{c\in\Lambda^k} \left(\begin{array}{c} \overleftarrow{\widetilde{\bigcirc}}(\neg\mathsf{rs}_c) \wedge \Box_{=\delta}(\mathsf{st} = s_j \Rightarrow \mathsf{rs}_c) \\ \vee \\ \overleftarrow{\widetilde{\bigcirc}}(\mathsf{rs}_c) \wedge \Box_{=\delta}(\mathsf{st} = s_j \Rightarrow \neg\mathsf{rs}_c) \end{array} \right) \right) \tag{16}$$

$$\triangle(\neg\mathsf{rs}_c, \mathsf{rs}_c) \quad \Rightarrow \quad \bigvee_k \left(\overleftarrow{\widetilde{\bigcirc}}\left(\mathsf{st} = s_i^k\right) \wedge \Box_{=\delta}\left(\mathsf{rs}_c \Rightarrow \mathsf{st} = s_j^k\right) \right) \tag{17}$$

Intuitively, the equivalence of (1) and (16) can be proved noting that, if $\triangle(\mathsf{st}, s_i, s_j)$ holds at instant t in some non-Berkeley behavior b because a transition $\langle s_i, s_j, \Lambda^k, \xi^k\rangle \in$

E is taken, then, for the non-Berkeleyness of b it must be $\mathsf{st} = s_j$ throughout $(t, t+\delta]$. As a consequence of (16), for any $c \in \Lambda^k$, if $\overleftarrow{\bigcirc}(\neg \mathsf{rs}_c)$ at t then rs_c holds at $t+\delta$. For the non-Berkeleyness of b it must be that $\overrightarrow{\bigcirc}(\mathsf{rs}_c)$ holds at t, hence $\triangle(\neg \mathsf{rs}_c, \mathsf{rs}_c)$ also holds at t. The case $\overleftarrow{\bigcirc}(\neg \mathsf{rs}_c)$ at t is handled in the same way. Similar reasoning can be used to prove the equivalence of (4) and (17).

Finally, formulas (16–17),(2–3),(5–6) yield the following over-approximations, after performing some discrete-time simplifications:

$$\mathrm{O}_\delta\,((16)) \equiv$$

$$\blacktriangle(\mathsf{st}, s_i, s_j) \Rightarrow \bigvee_k \left(\bigwedge_{c \in \Lambda^k} \begin{pmatrix} \mathrm{O}_\delta\left(\Xi(\xi^k)\right) \wedge \\ \overleftarrow{\Box}_{[0,1]}(\neg \mathsf{rs}_c) \wedge \Box_{[0,2]}(\mathsf{st} = s_j \Rightarrow \mathsf{rs}_c) \\ \vee \\ \overleftarrow{\Box}_{[0,1]}(\mathsf{rs}_c) \wedge \Box_{[0,2]}(\mathsf{st} = s_j \Rightarrow \neg \mathsf{rs}_c) \end{pmatrix} \right) \tag{18}$$

$$\mathrm{O}_\delta\,((17)) \equiv \blacktriangle(\neg \mathsf{rs}_c, \mathsf{rs}_c) \Rightarrow \bigvee_k \left(\overleftarrow{\Box}_{[0,1]}(\mathsf{st} = s_i^k) \wedge \Box_{[0,2]}(\mathsf{rs}_c \Rightarrow \mathsf{st} = s_j^k) \right) \tag{19}$$

$$\mathrm{O}_\delta\,((2)) \equiv \neg(\triangle(\mathsf{st}, s_i, s_j) \vee \blacktriangle(\mathsf{st}, s_i, s_j)) \tag{20}$$

$$\mathrm{O}_\delta\,((3)) \equiv (3) \tag{21}$$

$$\text{at } 0\text{:} \quad \mathrm{O}_\delta\,((5)) \equiv \bigwedge_{c \in C} \mathsf{rs}_c \wedge \Diamond_{=1} \left(\bigwedge_{c \in C} \neg \mathsf{rs}_c \right) \wedge \bigvee_{s_0 \in S_0} \Box_{[0,1]}(\mathsf{st} = s_0) \tag{22}$$

$$\mathrm{O}_\delta\,((6)) \equiv (6) \tag{23}$$

4.3 Summary

The following proposition, following from Propositions 1–2 and the results of the previous sections, summarizes the results of the discrete-time approximation formulas.

Proposition 3. *Let S be a real-time system described by TA $A = \langle \Sigma, S, S_0, \alpha, C, E \rangle$ and by a set of MTL specification formulas $\{\phi_j^{\mathsf{sys}}\}_j$ over items in $\mathcal{I}$ and propositions in $\mathcal{P}$. Also, let ϕ^{prop} be another MTL formula over items in $\mathcal{I} \cup \{\mathsf{st} : S, \mathsf{in} : \Sigma\}$ and propositions in $\mathcal{P} \cup R$. Then:*

– *if:*

$$\mathrm{Alw}\left(\phi^A_{(10)} \wedge \phi^A_{(11)} \wedge \phi^A_{(12)} \wedge \phi^A_{(13)} \wedge \phi^A_{(14)} \wedge \phi^A_{(15)} \wedge \bigwedge_j \Omega_\delta\left(\phi_j^{\mathsf{sys}}\right) \right) \Rightarrow \mathrm{Alw}(\mathrm{O}_\delta\left(\phi^{\mathsf{prop}}\right))$$

is $\mathbb{N}$-valid, then ϕ^{prop} is satisfied by all non-Berkeley runs $b \in \mathcal{B}_\chi^\delta$ of the system;

– *if:*

$$\mathrm{Alw}\left(\phi^{A}_{(18)} \wedge \phi^{A}_{(19)} \wedge \phi^{A}_{(20)} \wedge \phi^{A}_{(21)} \wedge \phi^{A}_{(22)} \wedge \phi^{A}_{(23)} \wedge \bigwedge_{j} \mathrm{O}_{\delta}\left(\phi_{j}^{\mathsf{sys}}\right)\right) \Rightarrow \mathrm{Alw}(\Omega_{\delta}\left(\phi^{\mathsf{prop}}\right))$$

is not $\mathbb{N}$*-valid, then* ϕ^{prop} *is false in some non-Berkeley run* $b \in \mathcal{B}^{\delta}_{\chi}$ *of the system.*

5 Implementation and Example

We implemented the verification technique of this paper as a plugin to the ℤot bounded satisfiability checker [17,18] called TAℤot. The plugin provides a set of primitives by which the user can define the description of a TA, of a set of MTL axioms, and a set of MTL properties to be verified. The tool then automatically builds the two discrete-time approximation formulas of Proposition 3. These are checked for validity over time $\mathbb{N}$; the results of the validity check allows one to infer the validity of the original dense-time models, according to Proposition 3.

The verification process in TAℤot consists of three sequential phases. First, the discrete-time MTL formulas of Proposition 3 are built and are translated into a propositional satisfiability (SAT) problem. Second, the SAT instance is put into conjunctive normal form (CNF), a standard input format for SAT solvers. Third, the CNF formula is fed to a SAT solving engine (such as MiniSat, zChaff, or MiraXT).

5.1 A Communication Protocol Example

We demonstrate the practical feasibility of our verification techniques by means of an example, where we verify certain properties of the following communication protocol.

Consider a server accepting requests from clients to perform a certain service (the exact nature of the service is irrelevant for our purposes). Initially, the server is *idle* in a passive open state. At any time, a client can initiate a protocol run; when this is the case, the server moves to a *try* state. Within T_1 time units, the state moves to a new s_1 state, characterizing the first request of the client for the service. The request can either terminate within T_2 time units, or time-out after T_2 time units have elapsed. When it terminates, it can do so either successfully (*ok*) or unsuccessfully (*ko*). In case of success, the protocol run is completed afterward, and the server goes back to being *idle*. In case of failure or time-out, the server moves to a new s_2 state for a second attempt. The second attempt is executed all similarly to the first one, with the only exception that the system goes back to the *idle* state afterward, regardless of the outcome (success, failure, or time-out). The timed automaton of Figure 1 models the protocol.[2]

We verified the following 5 properties of a single instance of the automaton:

[2] Since the definition of clock constraints forbids the introduction of exact constraints such as $A = T_2$, such constraints represent a shorthand for the valid clock constraint $T_2 \leq A < T + \delta$.

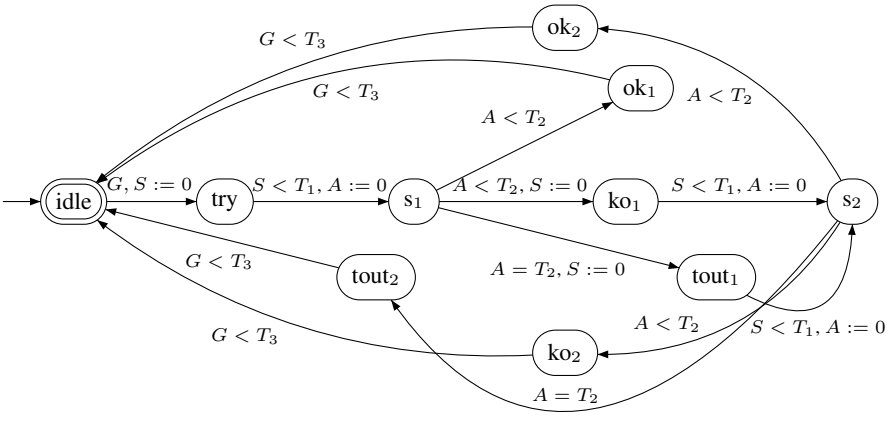


Fig. 1. Timed automaton modeling the communication protocol

1. "If there is a success, the server goes back to idle without passing through error states."
$$\mathrm{ok}_1 \vee \mathrm{ok}_2 \quad \Rightarrow \quad \mathsf{U}(\mathrm{ko}_1 \vee \mathrm{ko}_2, \mathrm{idle})$$
2. "If there is a failure, the server goes back to idle without passing through success states."
$$\mathrm{ko}_1 \vee \mathrm{ko}_2 \quad \Rightarrow \quad \mathsf{U}(\mathrm{ok}_1 \vee \mathrm{ok}_2, \mathrm{idle})$$
This property is false, and in fact counterexamples are produced in the tests.
3. "A full run of the protocol executes in no more than T_3 time units."
$$\mathrm{try} \quad \Rightarrow \quad \Diamond_{(0,T_3)}(\mathrm{idle})$$
This property cannot be verified due to the incompleteness of the method: whether a run is completed in T_3/δ time instants depends sensibly on how the sampling is chosen. However, if we slightly weaken the property by changing T_3 into $T_3+\delta$ the method is successful in verifying the property. In the tables, the (verified) property — modified in this way — is labeled 3'.
4. "The first attempt of the protocol is initiated no later than $2T_1 + T_2 + \delta$ time units after the run has been initiated."
$$\mathrm{s}_1 \quad \Rightarrow \quad \overleftarrow{\Diamond}_{(0,2T_1+T_2+\delta)}(\mathrm{try})$$
5. "A run is terminated within T_3 time units after a successful outcome, without going through failure states."
$$\mathrm{ok}_1 \quad \Rightarrow \quad \mathsf{U}_{(0,T_3)}(\neg(\mathrm{ko}_1 \vee \mathrm{ko}_2), \mathrm{idle})$$

We also considered concurrent runs of $N_r \geq 2$ instances of the automaton, synchronized under the assumption that two parallel protocol runs that are initiated concurrently

either both terminate successfully, or both terminate unsuccessfully. This is formalized by the following MTL formula:

$$\forall 1 \leq i < j \leq N_r : \quad \mathrm{try}^i \wedge \mathrm{try}^j \Rightarrow$$
$$\mathsf{U}\big(\neg(\mathrm{tout_2}^i \vee \mathrm{ko_2}^i), \mathrm{ok_1}^i \vee \mathrm{ok_2}^i\big) \wedge \mathsf{U}\big(\neg(\mathrm{tout_2}^j \vee \mathrm{ko_2}^j), \mathrm{ok_1}^j \vee \mathrm{ok_2}^j\big)$$
$$\vee$$
$$\mathsf{U}\big(\neg(\mathrm{ok_1}^i \vee \mathrm{ok_2}^i), \mathrm{tout_2}^i \vee \mathrm{ko_2}^i\big) \wedge \mathsf{U}\big(\neg(\mathrm{ok_1}^j \vee \mathrm{ok_2}^j), \mathrm{tout_2}^j \vee \mathrm{ko_2}^j\big)$$

Correspondingly, we introduce the following two properties to be verified in this concurrent system.

6. "If at some time one process succeeds and the other fails, then they have not begun the current run together."
$$\mathrm{ok_2}^A \wedge \mathrm{ko_2}^B \quad \Rightarrow \quad \mathsf{S}_{(0,T_3)}\big(\neg(\mathrm{try}^A \wedge \mathrm{try}^B), \mathrm{try}^A \vee \mathrm{try}^B\big)$$
7. "If at some time one process succeeds and the other failed recently, then they have not begun the current run together."
$$\mathrm{ok_2}^A \wedge \overleftarrow{\Diamond}_{(0,T_1)}\Big(\mathrm{ko_2}^B\Big) \quad \Rightarrow \quad \mathsf{S}_{(0,T_3)}\big(\neg(\mathrm{try}^A \wedge \mathrm{try}^B), \mathrm{try}^A \vee \mathrm{try}^B\big)$$

5.2 Experimental Evaluation

Tables 2 shows some results obtained in tests with TA$\mathbb{Z}$ot verifying the properties above. In all tests it is $\delta = 1$. For each test the table reports: the checked property; the number N_r of parallel protocol runs, according to which the discretizations are built; the values of other parameters in the model (i.e., T_1, T_2, T_3); the temporal bound k of the time domain (as $\mathbb{Z}$ot is a bounded satisfiability checker, it considers all the behaviors with period $\leq k$); the total amount of time and space (in MBytes) to perform each phase of the verification, namely formula building (FB), transformation into conjunctive normal form (CNF), and propositional satisfiability checking (SAT); and the total size (in thousands of clauses) of the propositional formulas that have been checked. The tests have been performed on a PC equipped with an AMD Athlon64 X2 Dual-Core Processor 4000+, 2 Gb of RAM, and Kubuntu GNU/Linux (kernel 2.6.22). TA$\mathbb{Z}$ot used GNU CLisp 2.41 and MiniSat 2.0 as SAT-solving engine.

The experiments clearly shows that the formula building time is usually negligible; the satisfiability checking time is also usually acceptably small, at least within the parameter range for the experiments we considered. On the contrary, the time to convert formulas in conjunctive normal form usually dominates in our tests. This indicates that there is significant room for practical scalability of our verification technique. In fact, from a computational complexity standpoint, the SAT phase is clearly the critical one, as it involves solving an NP-complete problem. On the other hand, the CNF routine has a quadratic running time.

Another straightforward optimization could be the implementation of the TA encoding directly in CNF, to bypass the `sat2cnf` routine. This can easily be done, because the structure of the formulas in the axiomatization is fixed. In conclusion, we can safely claim that the performances obtained in the tests are satisfactory in perspective, and they successfully demonstrate the practical feasibility of our verification technique.

Table 2. Checking properties of the communication protocol

PR.#	N_r	T_1, T_2, T_3	k	FB (time/mem)	CNF (time/mem)	SAT (time/mem)	# KCL.
1	1	3,6,18	30	0.1 min/114.6 Mb	3.9 min	0.3 min/90.2 Mb	520.2
2	1	3,6,18	30	0.1 min/228.6 Mb	7.8 min	0.5 min/180.1 Mb	1037.9
3	1	3,6,18	30	0.2 min/244.3 Mb	9.1 min	0.7 min/195.6 Mb	1112.4
3'	1	3,6,18	30	0.1 min/122.5 Mb	4.6 min	0.4 min/98.0 Mb	557.7
4	1	3,6,18	30	0.1 min/121.4 Mb	4.5 min	0.3 min/97.4 Mb	553.2
5	1	3,6,18	30	0.1 min/122.6 Mb	4.6 min	0.4 min/97.9 Mb	557.3
1	1	3,6,24	36	0.1 min/146.8 Mb	6.3 min	0.5 min/117.9 Mb	669.1
2	1	3,6,24	36	0.2 min/292.9 Mb	12.5 min	0.9 min/235.4 Mb	1335.2
3	1	3,6,24	36	0.2 min/319.0 Mb	15.4 min	1.2 min/258.6 Mb	1459.0
3'	1	3,6,24	36	0.1 min/159.9 Mb	7.6 min	0.7 min/129.3 Mb	731.3
4	1	3,6,24	36	0.1 min/155.0 Mb	7.2 min	0.5 min/126.4 Mb	708.5
5	1	3,6,24	36	0.1 min/160.3 Mb	7.8 min	0.9 min/129.8 Mb	731.3
1	1	4,8,24	40	0.1 min/171.9 Mb	8.5 min	0.7 min/136.2 Mb	785.5
2	1	4,8,24	40	0.2 min/343.1 Mb	17.2 min	1.2 min/271.9 Mb	1567.7
3	1	4,8,24	40	0.3 min/372.1 Mb	21.0 min	1.7 min/297.3 Mb	1705.1
3'	1	4,8,24	40	0.1 min/186.5 Mb	10.2 min	0.9 min/148.9 Mb	854.6
4	1	4,8,24	40	0.1 min/184.6 Mb	10.3 min	0.8 min/148.3 Mb	846.6
5	1	4,8,24	40	0.1 min/186.9 Mb	10.4 min	1.1 min/148.9 Mb	854.5
1	1	3,15,90	105	2.2 min/819.6 Mb	203.8 min	20.0 min/674.7 Mb	3826.9
2	1	3,15,90	105	4.4 min/1637.3 Mb	389.2 min	31.3 min/1352.5 Mb	7645.2
3	1	3,15,90	105	5.6 min/1945.7 Mb	561.2 min	61.1 min/821.2 Mb	9103.8
3'	1	3,15,90	105	2.9 min/974.0 Mb	286.7 min	61.1 min/410.9 Mb	4557.2
4	1	3,15,90	105	2.3 min/864.5 Mb	224.8 min	14.4 min/381.0 Mb	4042.8
5	1	3,15,90	105	3.2 min/981.1 Mb	291.4 min	342.5 min/463.4 Mb	4571.0
6	2	3,6,18	30	0.2 min/241.6 Mb	16.7 min	1.6 min/192.4 Mb	1098.9
7	2	3,6,18	30	0.2 min/244.9 Mb	17.3 min	1.8 min/194.4 Mb	1114.4
6	2	3,6,24	36	0.2 min/313.7 Mb	28.7 min	2.4 min/254.5 Mb	1432.0
7	2	3,6,24	36	0.2 min/317.6 Mb	31.0 min	2.7 min/257.5 Mb	1450.5
6	2	4,8,24	40	0.3 min/366.3 Mb	39.5 min	3.5 min/294.1 Mb	1675.3
7	2	4,8,24	40	0.3 min/371.5 Mb	38.2 min	3.8 min/297.0 Mb	1700.1
6	4	3,6,18	30	0.3 min/472.3 Mb	61.4 min	5.0 min/377.3 Mb	2145.6
7	4	3,6,18	30	0.3 min/475.5 Mb	62.3 min	5.3 min/379.3 Mb	2161.1
6	4	3,6,24	36	0.5 min/609.3 Mb	101.6 min	8.7 min/483.6 Mb	2777.7
7	4	3,6,24	36	0.5 min/613.2 Mb	103.1 min	9.2 min/486.2 Mb	2796.2
6	4	4,8,24	40	0.5 min/712.3 Mb	139.2 min	12.1 min/577.0 Mb	3254.6
7	4	4,8,24	40	0.6 min/717.5 Mb	141.0 min	12.6 min/580.3 Mb	3279.5

6 Conclusion

In this paper, we introduced a technique to perform partial verification of real-time systems modeled with dense time and using mixed operational and descriptive components. The proposed approach is fully automated and implemented on top of a discrete-time bounded satisfiability checker. We experimented with a non-trivial example of a communication protocol, where concurrent runs of the protocol are synchronized

through additional MTL formulas, hence building a mixed model. Verification tests showed consistent results and reasonable performances. As future work, we intend to improve the efficiency of the technique by using a pure CNF encoding, and to investigate the use of other operational formalisms, such as timed Petri nets.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R., Feder, T., Henzinger, T.A.: The benefits of relaxing punctuality. Journal of the ACM 43(1), 116–146 (1996)
3. Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992)
4. Alur, R., Henzinger, T.A.: Real-time logics: Complexity and expressiveness. Information and Computation 104(1), 35–77 (1993)
5. De Wulf, M., Doyen, L., Raskin, J.-F.: Almost ASAP semantics: from timed models to timed implementations. Formal Aspects of Computing 17(3), 319–341 (2005)
6. Fainekos, G.E., Pappas, G.J.: Robust sampling for MITL specifications. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, Springer, Heidelberg (2007)
7. Furia, C.A.: Scaling up the formal analysis of real-time systems. PhD thesis, DEI, Politecnico di Milano (May 2007)
8. Furia, C.A., Mandrioli, D., Morzenti, A., Rossi, M.: Modeling time in computing. Technical Report 2007.22, DEI, Politecnico di Milano (January 2007)
9. Furia, C.A., Pradella, M., Rossi, M.: Automated verification of dense-time MTL specifications via discrete-time approximation. In: Cuellar, J., Maibaum, T.S.E. (eds.) FM 2008. LNCS, vol. 5014, pp. 132–147. Springer, Heidelberg (2008)
10. Furia, C.A., Pradella, M., Rossi, M.: Practical automated partial verification of multi-paradigm real-time models (April 2008), http://arxiv.org/abs/0804.4383
11. Furia, C.A., Rossi, M.: Integrating discrete- and continuous-time metric temporal logics through sampling. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 215–229. Springer, Heidelberg (2006)
12. 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)
13. Henzinger, T.A., Raskin, J.-F., Schobbens, P.-Y.: The regular real-time languages. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 580–591. Springer, Heidelberg (1998)
14. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4), 255–299 (1990)
15. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1–2) (1997)
16. Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
17. Pradella, M.: ℤot (March 2007), http://home.dei.polimi.it/pradella
18. Pradella, M., Morzenti, A., San Pietro, P.: The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In: Proc. of ESEC/FSE 2007 (2007)

Specifying and Verifying Sensor Networks: An Experiment of Formal Methods

Jin Song Dong[1], Jing Sun[2], Jun Sun[1,⋆], Kenji Taguchi[3], and Xian Zhang[1]

[1] School of Computing,
National University of Singapore
Tel.: +65 6516 4244; Fax: +65 6779 1610
{dongjs,sunj,zhangxi5}@comp.nus.edu.sg
[2] Department of Computer Science,
The University of Auckland
j.sun@cs.auckland.ac.nz
[3] Information Systems Architecture Research Division,
Grace Center, National Institute of Informatics

Abstract. With the development of sensor technology and electronic miniaturization, wireless sensor networks have shown a wide range of promising applications as well as challenges. Early stage sensor network analysis is critical, which allows us to reveal design errors before sensor deployment. Due to their distinguishable features, system specification and verification of sensor networks are highly non-trivial tasks. On the other hand, numerous formal theories and analysis tools have been developed in formal methods community, which may offer a systematic method for formal analysis of sensor networks. This paper presents our attempt on applying formal methods to sensor network specification/verification. An integrated notation named *Active Sensor Processes* is proposed for high-level specification. Next, we experiment formal verification techniques to reveal design flaws in sensor network applications.

1 Introduction

With the development of sensor technology and electronic miniaturization, sensor integration makes it possible to produce extremely inexpensive sensing devices. The sensors have been equipped with significant processing, memory, and wireless communication capabilities. Thus, they are capable of performing complex in-network computation besides sensing and communication. Wireless sensor networks have shown a wide range of promising applications in a variety of domains [12], e.g., environmental monitoring, acoustic detection, smart spaces, inventory tracking, etc.

In recent years, a number of sensor devices and sensor programming languages have been actively developing [1,4,20]. In the following, we review several features of ad hoc and sensor networks which distinguish them from ordinary systems. In general, sensor networks may be categorized as *event-based distributed reactive hybrid* systems.

⋆ Corresponding author.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 318–337, 2008.

- The nature of sensors is event-based. Sensors' behaviors are often stated in terms of how they respond to internal/external events. Sensors may have continuous interfaces for sensing/actuating as well as discrete message input/output for inter-sensor communication.
- Sensor networks may be re-configurable, i.e., part of the behaviors may be updated dynamically. Deployed sensors may need to be updated with new programs to cope with new system requirements or reused for completely new tasks.
- Sensor network applications are unlikely interested in the state of an individual sensor. Rather, applications focus on the data generated by sensors. Sensor network nodes are data-centric. There may not be a unique global address (like IP in the Internet) associated with each sensor. The data generated at a sensor node is named by attributes and applications request data matching certain attribute values. Date-centric routing is favored over end-to-end routing.
- Sensor networks are application-aware. Traditional networks are designed to accommodate a wide variety of applications. Sensor networks are however application-specific, i.e., they are configured and deployed for a specific application. Thus, they are designed with the knowledge of the types of sensors, the geography, the data format generated by the sensors, etc.

On one hand, early stage analysis of sensor networks is critical as, once deployed, sensors may not be easily accessible and updated with corrections. The unique features of wireless sensor networks present unique challenges for specification, verification and synthesis. On the other hand, numerous formal theories (e.g., broadcasting messages [25], higher-order processes [30], mobile processes [5]) and tools (e.g., UPPAAL, SPIN) have been developed. We believe that wireless sensor networks are a fruitful application domain of formal methods, which shall provide methodologies as well as tools for systematic sensor networks specification and verification. This paper presents our initial attempt on applying formal methods to sensor networks.

In order to apply the rich set of formal methods and theories, the very first task is to construct a formal description of sensor networks. A formal system description requires construction of a high-level mathematical model of the system, which can later be used for a variety of system analysis tasks such as simulation, verification, performance evaluation and synthesis. In this paper, we adapt ideas from multiple existing formal specification languages [23,25,6] and propose a simple integrated notation for formal sensor network specification, namely *Active Sensor Processes* (shortened as ASP). ASP is based on classic process algebras with extensions solely for hybrid broadcasting systems. Next, we demonstrate how to verify sensor network applications using state-of-the-art verification techniques. Verification based on ASP (instead of concrete implementation) allows us to focus on the key aspects of the application without being disturbed by irrelevant details. For instance, it is desirable to prove the soundness of the high-level specification of a wireless routing protocol (i.e., a package will reach the destination provided that it is feasible) assuming reliable a link layer protocol. Performed early in the design stage, such modeling and analysis offers the promise of a systematic approach. We show that using existing system analysis tools, previously unknown design flaw can be revealed. Nonetheless, we show that systematic sensor network verification may require verification capability beyond existing techniques and tools.

As for related works, the research on sensor networks has been influenced by both the traditional network community and the database community. The former group tends to focus on finding efficient communicating protocol [32,17]. The latter focuses on in-network data aggregation and data querying [8,22]. Other works include building sensor hardwares and their middleware support [20]. There have also been proposals for domain specific languages which present programming models for writing sensor network programs [4,22]. We believe formal methods community can contribute to the development of sensor network systems by providing new modeling and design techniques capturing high level system behaviors. To our best knowledge, there have been few formal languages proposed for generic sensor network modeling [24,31]. The previous closest to ASP is the notion of SensorML [3], which offers a modeling language for sensor processes. Comparing to SensorML, our language models the dynamic behaviors of sensor networks. ASP has a formal semantics, which is essential for formal analysis. For instance, by translating a large subset of ASP to equivalent timed automata, we may reuse existing model checkers for formal verification. There have been many languages proposed for modeling hybrid or mobile reactive systems, e.g., Ambients, KLAIM, TCOZ, various extensions to π-calculus or automata, etc. However, applying existing formalisms may not be possible or optimal because of the unique features of sensors. For instance, the classic CSP, CCS, π-calculus and Ambient calculus rely on single communication mechanism (e.g., lock-step synchronization between processes), which is not suitable for inter-sensor communication. Naturally, there are different ways of communication in sensor networks, i.e., the sensing/actuating may be continuous rather then discrete; the messaging between sensor nodes are asynchronous as well as broadcasting; and there may be lock-step synchronization between processes running in parallel which reside at the same sensor node. Moreover, the network topology in sensor networks is highly dynamic, which depends on a lot on the geographic location of the sensor and its own characteristics like radio range. Existing formal specification languages, like π-calculus and Ambient calculus, support network dynamic reconfiguration by explicitly changing the channel names or the residing location of a process and thus are in-effective for modeling data-centric sensor network routing. Nonetheless, we believe wireless sensor networks are a fruitful domain for the scattered works on theoretical development on communication as well as process semantics (e.g., broadcasting semantics, higher-order processes and others as evidenced in [24,25]).

The remainder of the paper is organized as follows. Section 2 explains the constructs of ASP using illustrative examples. Section 3 explains the semantics of the language. Section 4.1 models and verifies (using existing tools) a network code propagation algorithm. Section 4.2 studies modeling and verification of a class of sensor network routing protocols. Section 5 concludes the paper and reviews related works and future works.

2 Specifying Sensor Networks

The objective is to construct a concise and precise model of the sensor nodes. In order to facilitate later system analysis, the language is designed to be lightweight. In this section, we present the notation named *Active Sensor Processes*. We remark that ASP

$$
\begin{array}{lll}
P ::= & Stop \mid Skip & \text{– deadlock and termination} \\
\mid & Idle(d) & \text{– delay} \\
\mid & e \rightarrow P & \text{– event-prefix} \\
\mid & x := exp \rightarrow P & \text{– assignment} \\
\mid & P;\ Q & \text{– sequential composition} \\
\mid & P \,\Box\, Q & \text{– choice} \\
\mid & P \lhd b \rhd Q & \text{– conditional branching} \\
\mid & P \,\triangledown\, Q & \text{– interrupt} \\
\mid & P \,|[\, X \,]|\, Q & \text{– local synchronous composition} \\
\mid & s \odot x \rightarrow P & \text{– sensing} \\
\mid & s \otimes v \rightarrow P & \text{– actuating} \\
\mid & c?X \rightarrow P & \text{– inter-sensor message input} \\
\mid & c!X \rightarrow P & \text{– inter-sensor message output} \\
\mid & P = Q & \text{– process referencing}
\end{array}
$$

Fig. 1. ASP Process Syntax

adapts features from Timed CSP [27] and TCOZ [23] and then extends it with language constructs which are dedicated to sensor networks.

For simplicity, we distinguish three types, i.e., the set of all data values $\mathcal{V}$, the set of all events $\mathcal{E}$ and the set of all processes $\mathcal{P}$. The specification of a sensor node contains two parts. One is the mapping of a set of data variables to their values. The other is the mapping of a set of process variables to process expressions. Unlike traditional CSP/CCS, processes are allowed to be re-defined dynamically so as to model sensors which are designed to be reconfigurable. This is achieved by assigning a process expression to process names dynamically, in the same way new values are assigned to data variables. Data variables identify the context of the node, which may be categorized into two groups. The first group is a set of pre-defined control variables which determines the network topology. For instance, one way to capture the network topology is through two predefined variables, namely $location$ and $range$. Variable $location$ identifies the location of the sensor node. Variable $range$ identifies its radio range, which in terms identifies all connected sensor nodes together with the variable $location$. The second group is a set of local data variables, e.g., control variables, sensed data store, etc. A process expression is formed by the syntax summarized in Figure 1. In the following, we will briefly review the process constructs using illustrative examples. A number of process constructs introduced in CSP and Timed CSP are reused. Note that some have different semantics.

The process $Skip$ terminates successfully. The process $Stop$ deadlocks. A sensor node which behaves as $Stop$ (if the battery runs out or the antenna is broken) disappears from the network since it can not communicate with the rest of the system any more. The process $Idle(d)$ where d is a positive real number idles for exactly d time units. A sensor node behaving as $Idle(d)$ may go to sleep mode or power off (to save energy). In contrast to sensors which behave as $Stop$, the sensor wakes up after exactly d time units or responding to some interrupts. Note that clocks, as data variables, are always local to a sensor node. The process $e \rightarrow P$ is called event-prefixing. It engages e and then behaves as P, where e is an abstract event. If the event is shared by multiple

processes running in parallel residing at the same sensor node, the event acts like a synchronization barrier. Local variables or processes may be updated by assignments of the form $x := exp$ where exp is an expression.

Diversity of behaviors are specified using choices, which are often guarded with events or Boolean conditions. Process $a \rightarrow P \ \Box \ b \rightarrow Q$ will proceed as specified by P if a is firstly engaged. A conditional branch is written as $P \lhd b \rhd Q$, where b is a Boolean expression over the sensor's context. It behaves as specified by P if b evaluates to true, otherwise it behaves as specified by Q. We view interrupt as a biased choice. The process $P \bigtriangledown e \rightarrow Q$ behaves as P until the moment e is engaged and then behaves as Q afterward. Alphabetized parallel composition is written as $P \,|[\, X \,]|\, Q$, where X is a set of events. Events in X must be synchronized by P and Q. X is omitted if it is exactly the set of common events of P and Q. Recursion is defined through process referencing. Its semantics is defined as Tarski's weakest fixpoint as in CSP [16].

Sensor nodes may communicate with its local continuous environment via sensing or actuating. A sensor node can sense data from its ever changing environment so as to detect certain phenomenon or to be aware of its context. The process $s \odot x \rightarrow P$ reads the value x from a sensing channel s from the environment. If the external environment is specified as a continuous function, the object 'monitors' the value of the continuous function through s. Sensor nodes with the same sensing device may receive the same datum from the external environment. There may be multiple sensing channels on one sensor node. Different sensing channels may be dedicated to different phenomenons. The data sensed by a sensor node may come from the external environment or another sensor node in the system. Sensor nodes influence its environment through actuating. Process $s \otimes v \rightarrow P$ actuates value v continuously to the environment. A sensing channel and an actuating channel match if they share the channel name.

Example 1. A light sensor of a camera detects the light condition and outputs the light level continuously. The camera reads the light level regularly so as to update the screen[1].

LightSensor

$$Main = Idle(2);\ daylight \odot x \rightarrow illumination \otimes x \rightarrow Main$$

where $daylight$ is a sensing channel and x is the sensed value. The process $Main$ identifies the behaviors of the object after initialization (as in TCOZ [23]). After sensing the day light level from the environment, its value is actuated on channel $illumination$. The process repeats after 2 time units.

Display

$$value = 0$$
$$Main = illumination \odot x \rightarrow (Main \lhd value = x \rhd refresh \rightarrow value := x \rightarrow Main)$$

where $value$ is a data variable recording the current light level. Its initial value is 0. $refresh$ is the event of refreshing the screen. The above defines a local display device.

[1] We use a Z-schema like syntax to group components of a sensor node. It by no means implies that we adapt the Z syntax and semantics.

It reads the value of $illumination$ through sensing. If the value is different from the previous one, the screen is refreshed to show the new value, else the screen is not refreshed. Thus, the illumination displayed is in real-time. Initially, 0 is displayed. Channel $daylight$ connects the system with the external environment as there is no matching sensing channel named $daylight$. □

As shown above, the specification of one sensor node contains a set of mapping from data/process variables to their value. The initial values identify the initial state. The process $Main$ identifies the behavior of the node. Sensor nodes which form a network must communicate with each other, e.g., typically through radio transmission. There are unique characteristics about inter-sensor messaging. First of all, messaging between different nodes is almost always asynchronous because processing time per bit communicated is plentiful in sensor networks, i.e., CPUs are fast and bandwidths are low. Depending on the communication media and the transfer rate, it may take considerably long time. Secondly, there may not be a global identity for each sensor and thus end-to-end communication is unlikely in sensor networks. Thus, broadcasting is favored in sensor networks instead of messaging through channels shared by a pre-determined set of processes (as in CSP [16]).

$c!X \rightarrow P$ – output
$c?X \rightarrow P$ – input

where c is a channel. The message content X, which is called a $gradient$, could be an abstract event or a data message or even a process itself. We assume that the output is broadcasted so that all sensors within the range may receive it. Yet only those intended ones may process it. A receiving sensor node may only listen on channels of its own interest. A $gradient$ is always treated as a single parameter, and its syntax and how it is evaluated should be defined by a designer of the system and be incorporated into the operational semantics.

Example 2. Suppose the sensors have been deployed around a volcano (e.g., they are thrown from an aircraft) so as to monitor the volcano activity. The sensors report through radio whenever the sensed temperature is above certain threshold. Due to the rising of temperature, we now want those sensors which sensed high temperature to report more frequently. Thus, a station is set up near the volcano, which is modeled as follows,

Station
$Main = c!set(5, 20) \rightarrow Main$

where c is a channel, set is a message tag and $(5, 20)$ is a compound message in a pre-agreed format. The first number identifies the intended receivers and the second is used to update the frequency. A channel can be implemented on a separate port or radio frequency or as simple as a flag bit in the message package. The station repeatedly broadcasts the message. The sensors are designed as follows,

Node

$$
\begin{array}{l}
delay = 5 \\
data = 0 \\
Routine = Idle(delay);\ temperature \odot x \rightarrow data := x \rightarrow Routine \\
Update = c?set(threshold, x) \rightarrow (delay := x \rightarrow Update \\
\qquad \lhd\ data > threshold \rhd\ Update) \\
Main = Routine \parallel Update
\end{array}
$$

where $delay$ is a local variable used to control sensing frequency and $data$ is used to store the most recent sensed data. The $Main$ process is composed two sub-processes running in parallel. Process $Routine$ executes a routine task, i.e., after idling for $delay$ time units, collects temperature data from the environment and then store the sensed data into variable $data$. Process $Update$ awaits for messages from the station. Once a message arrives on channel c, if the newly sensed data is above the $threshold$, the $newdelay$ is adapted. Otherwise, process $Update$ is repeated.

The above design relies on a pre-defined message format, which is acceptable since sensor networks may be application specific. Alternatively, instead of updating one parameter of a process in a pre-determined way, a flexible sensor may be designed to be reconfigurable so that its behavior can be changed dynamically to cope with different tasks. The following shows such an approach based on the notion of higher-order processes, i.e., a message can be a process itself.

ReconStation

$$
Sender = c!set(110, Idle(20)) \rightarrow Sender
$$

The $sender$ sends a piece of program, i.e., $Idle(20)$, to directly change the behaviors of the sensors. A re-configurable sensor is designed as follows,

ReconNode

$$
\begin{array}{l}
data = 0 \\
Recon = Idle(5) \\
Routine = Recon;\ temperature \odot x \rightarrow data := x \rightarrow Routine \\
Update = c?set(threshold, X) \rightarrow \\
\qquad (Recon := X \rightarrow Update \lhd\ data > threshold \rhd\ Update) \\
Main = Routine \parallel Update
\end{array}
$$

where $Recon$ is defined as a process variable. $Routine$ is to execute $Recon$ first followed by sensing. Notice that whenever a process reference (e.g., $Recon$) is invoked, it is replaced by its current value. Once a new piece of program is received, the value of $Recon$ is updated by an assignment. □

There are two levels of concurrency in sensor networks. Firstly, local processes of a sensor node or computation devices connected by wire where communication delay is ignored can execute concurrently with possible barrier synchronization. Secondly, different sensor nodes run independently with each other and communicate through

message passing. The parallel composition of local processes located at the same sensor node is denoted as $P \,||[\, X \,]||\, Q$ where common events of Σ_P and Σ_Q are synchronized. All sensor nodes are implicitly executing in parallel and the communication between different sensor nodes is through asynchronous message passing[2].

Example 3. Because inter-sensor messaging may take considerably long time, it is often desirable to have separate processes for local computation and interfacing so that local computation carries on without being delayed by message sending or receiving. The following shows such a design,

SampleNode

$$
\begin{array}{l}
average = 0 \\
n = 0 \\
Accumulate = data \odot x \rightarrow average := (average \times n + x)/(n+1) \rightarrow \\
\qquad\qquad n := n + 1 \rightarrow Idle(2);\ Accumulate \\
Interface = sink!average \rightarrow Interface \\
Main = Interface \parallel Accumulate
\end{array}
$$

where n is a counter. Process $Interface$ handles communication (with a data sink). It repeatedly reads the $average$ and then sends it out. Process $Accumulate$ collects data from the environment. It gets the value of $data$ through sensing. It then computes the new average. After idling for 2 time units, process $Accumulate$ is invoked again. □

Complex process constructs can be composed from the simple ones. For instance, a process that times out after some time units is written as $P \triangleright_d Q$. The process Q takes control from the process P if P has not made a move after d time units elapsed, i.e., $P \triangleright_d Q$ is equivalent to $P \,\Box\, (Idle(d);\ Q)$. A process that is interrupted after executing exactly d time units is written as $P \bigtriangledown_d Q$. The process Q takes control from P after d time units, i.e., $P \bigtriangledown_d Q$ is equivalent to $P \bigtriangledown (Idle(d);\ Q)$. $[b] \bullet P$ is a guarded process which behaves as P when b is evaluated to true, i.e., $[b] \bullet P \mathrel{\widehat{=}} P \lhd b \rhd Stop$. Since they are considered as syntax sugars, we skip the rest for brevity.

3 Operational Semantics

In this section, the operational semantics of ASP is explained. The configuration of a node is composed of two components, i.e., the current process expression and the binding of data/process variables to their current values. Given a set of sensor nodes, the local state of each sensor node constitutes the global configuration.

Let B be a binding, which maps data variables to a data values or process variables to process expressions. Let P be a process. A configuration of a node is a pair of the form (P, B). The global state of a network of n nodes is $\{(P_1, B_1), (P2, B_2), \cdots, (P_n, B_n)\}$, i.e., the configuration of all sensor nodes. For simplicity, we write $(P_1, B_1) \;|||\; (P_2, B_2)$ to denote that (P_1, B_1) and (P_2, B_2) are part of

[2] Nearby sensors may interact through sensing/actuating.

the global state[3], i.e., there are a node with configuration (P_1, B_1) and a node with configuration (P_2, B_2) running in parallel. In the previous examples, the initialization consists of a set of equations of the form $name = initial\_value$. The initial state of a node is $(Main, BInit)$ where $Main$ is the main process and $BInit$ is the binding of all variables with their initial values. The following action prefixes are defined as abbreviations.

$$\begin{array}{l} \alpha ::= s \odot v \mid s \otimes v \\ \beta ::= c?v \mid c!v \\ \lambda ::= e \mid \tau \mid \checkmark \\ a ::= \alpha \mid \lambda \end{array}$$

where τ is the event of idling and $\checkmark$ is the event of termination. The operational semantics is presented using a set of transition rules associated with the language constructs. A transition is of the form $(P, B) \overset{a}{\hookrightarrow} (P', B')$, which denotes that a process P with a binding B performs an action a and evolves to a new process P' and a new binding B'. A new binding of a variable x to v in B is denoted by $B \oplus \{x \mapsto v\}$, i.e., the old value of x is replaced by v.

Inspired by the operational semantics defined in [26] for Timed CSP, we extend the rules with a component B to cope with our setting, as presented in Appendix A. We remark that $\Box$, $|[\, X \,]|$ and $|||$ are symmetric. Because of the shared variables, rules in [26] are extended to capture semantics of operators which deal with variables. Figure 2 presents those transitions rules. Rule $Assign$ states that an assignment replaces the value of variable x with $eval(B, exp)$ which is the value of exp evaluated against binding B. We remark that x may be a process variable and exp may be a process expression. Rule $Con1$ and $Con2$ capture the semantics of conditional branching. If the the condition b is true, written as $B \models b$, the system proceeds as P. Otherwise, it behaves as Q. Timing information is important in modeling (and verifying) sensor networks. We adapt a simple explicit-time approach, i.e., a variable $time$ is used to represent the current time [18]. Rule $Idle1$ and $Idle2$ captures how process $Idle(d)$ behaves. Notice that the behaviors of the clock are modeled implicitly as a process which updates the variable $time$. Rule $ProcDef$ deals with process referencing. The idea is to load the definition of P dynamically from B.

In order to capture the semantics of sensing/actuating, we introduce discard actions. Discard actions are defined as $\alpha :$. The discard action $(P, B) \overset{\alpha:}{\hookrightarrow} (P, B)$ means that P discards the action α. Figure 3 shows rules associated with sensing and actuating. These are based on CBS (Calculus of Broadcasting Systems) in [25] except rule $DiscardSensing$. Rule $DiscardSensing$ means that any node with the same sensor name may not receive the data from the environment or some corresponding actuator. This rule can mimic the locality of sensors. We regard the set of rules which includes *Sensing*, *DiscardSensing*, *SAParallel* and other *discard* rules (which only take the discard sensing action) the sensor process calculus, which captures the behaviors of sensors receiving data only from the environment. This set of rules is semantically weak but models how sensors behave in general.

[3] The operator $|||$ denotes interleaving in CSP. Here it implies there is no barrier synchronization among different sensor nodes.

$$\frac{eval(B, exp) = v}{(x := exp \rightarrow P, B) \overset{e}{\hookrightarrow} (P, B \oplus \{x \mapsto v\})} [\ Assign\ ]$$

$$\frac{B \models b \qquad (P, B) \overset{\lambda}{\hookrightarrow} (P', B')}{(P \lhd b \rhd Q, B) \overset{\lambda}{\hookrightarrow} (P', B')} [\ Con1\ ] \qquad \frac{B \not\models b \qquad (Q, B) \overset{\lambda}{\hookrightarrow} (Q', B')}{(P \lhd b \rhd Q, B) \overset{\lambda}{\hookrightarrow} (Q', B')} [\ Con2\ ]$$

$$\frac{\{time \mapsto n\} \subseteq B \qquad d > m}{(Idle(d), B) \overset{\tau}{\hookrightarrow} (Idle(d - m), B \oplus \{time \mapsto n + m\})} [\ Idle1\ ]$$

$$\frac{\{time \mapsto n\} \subseteq B}{(Idle(d), B) \overset{\checkmark}{\hookrightarrow} (Stop, B \oplus \{time \mapsto n + d\})} [\ Idle2\ ]$$

$$\frac{\{P \mapsto Q\} \subseteq B \qquad (Q, B) \overset{\lambda}{\hookrightarrow} (Q', B')}{(P, B) \overset{\lambda}{\hookrightarrow} (Q', B')} [\ ProcInst\ ]$$

$$\frac{P = Q \qquad (Q, B) \overset{\lambda}{\hookrightarrow} (Q', B')}{(P, B) \overset{\lambda}{\hookrightarrow} (Q', B')} [\ ProcDef\ ]$$

Fig. 2. Basic Operational Rules

Similarly, we define the rules for inter-sensor communication. All the rules for inter-sensor messaging, presented in Appendix B, are based on CBS but we need to take care of *gradient*. The syntax of gradients and how they will be evaluated should be pre-defined by a specifier of a sensor network system.

4 Case Studies

In this section, we demonstrate how ASP can be applied to model real-world sensor network applications concisely. Next, we discuss sensor network verification based on the ASP models. We show that using existing model checkers, bugs which are not previously known may be detected. Yet existing verification tools and techniques may not be sufficient in general.

4.1 The Trickle Algorithm

Communication in sensor networks may be extremely costly (in terms of time and battery). For some applications, sending a data of tens of kilobytes can have the same cost as days of operation. Thus, code propagation by flooding is undesirable. The Trickle algorithm [21] is a self-regulating algorithm for code (or large datum) propagation and

$$\frac{}{(s \odot x \to P, B) \overset{s\odot v}{\hookrightarrow} (P, B \oplus \{x \mapsto v\})} \; [\; Sensing \;]$$

$$\frac{}{(s \otimes v \to P, B) \overset{s\otimes v}{\hookrightarrow} (P, B)} \; [\; Actuating \;]$$

$$\frac{}{(s \otimes v \to P, B) \overset{s\otimes v:}{\hookrightarrow}} \; [\; DiscardActuating \;]$$

$$\frac{}{(s \odot x \to P, B) \overset{s\odot v:}{\hookrightarrow}} \; [\; DiscardSensing \;]$$

$$\frac{(P, B_1) \overset{s\odot v}{\hookrightarrow} (P', B_1') \quad (Q, B_2) \overset{s\otimes v}{\hookrightarrow} (Q', B_2)}{(P, B_1) \;|||\; (Q, B_2) \overset{s\otimes v}{\hookrightarrow} (P', B_1') \;|||\; (Q', B_2)} \; [\; SACom(1) \;]$$

$$\frac{(P, B_1) \overset{s\otimes v}{\hookrightarrow} (P', B_1) \quad (Q, B_2) \overset{s\odot v}{\hookrightarrow} (Q', B_2')}{(P, B_1) \;|||\; (Q, B_2) \overset{s\otimes v}{\hookrightarrow} (P', B_1) \;|||\; (Q', B_2')} \; [\; SACom(2) \;]$$

$$\frac{(P, B_1) \overset{s\odot v}{\hookrightarrow} (P', B_1') \quad (Q, B_2) \overset{s\odot v}{\hookrightarrow} (Q', B_2')}{(P, B_1) \;|||\; (Q, B_2) \overset{s\odot v}{\hookrightarrow} (P', B_1') \;|||\; (Q', B_2')} \; [\; SAParallel \;]$$

$$\frac{(P, B_1) \overset{\alpha:}{\hookrightarrow} \quad (Q, B_2) \overset{\alpha:}{\hookrightarrow}}{(P, B_1) \;|||\; (Q, B_2) \overset{\alpha:}{\hookrightarrow}} \; [\; SAJoinDiscard \;]$$

$$\frac{(P, B_1) \overset{\alpha:}{\hookrightarrow} \quad (Q, B_2) \overset{\alpha}{\hookrightarrow} (Q', B_2')}{(P, B_1) \;|||\; (Q, B_2) \overset{\alpha}{\hookrightarrow} (P, B_1) \;|||\; (Q', B_2')} \; [\; SADiscard(1) \;]$$

$$\frac{(P, B_1) \overset{\alpha}{\hookrightarrow} (P', B_1') \quad (Q, B_2) \overset{\alpha:}{\hookrightarrow}}{(P, B_1) \;|||\; (Q, B_2) \overset{\alpha}{\hookrightarrow} (P', B_1') \;|||\; (Q, B_2)} \; [\; SADiscard(2) \;]$$

Fig. 3. Sensor-Actuator rules

maintenance in wireless sensor networks. It combines typical sensor network behaviors like broadcasting, higher-order processes, etc. The sensors are designed to be reconfigurable, i.e., the received code will be installed and executed. It uses a "polite gossip" policy. Each node announces its metadata (e.g., a version number) every few

Mote

$$
\begin{aligned}
&data = 0\\
&version = 0\\
&counter = 0\\
&threshold = 2\\
&timer = 2\\
&tau = 5\\
&Routine = sense \odot x \rightarrow data := x;\ Idle(5);\ Routine\\
&Update = c?pro(v, R) \rightarrow ((version := v;\ Routine := R;\ Update)\\
&\qquad \lhd\ version < v \rhd\ Update)\\
&Gossip = (Talk \parallel Reply) \bigtriangledown_{tau}\ counter := 0;\ Gossip\\
&Talk = Idle(timer);\ (meta!version \rightarrow Talk \lhd\ counter < threshold \rhd\ Talk)\\
&Reply = meta?x \rightarrow ([x = version] \bullet counter := counter + 1;\ Reply\ \Box\\
&\qquad [x > version] \bullet meta!version \rightarrow Reply\ \Box\\
&\qquad [x < version] \bullet c!pro(version, Routine) \rightarrow Reply)\\
&Main = Routine \parallel Update \parallel Gossip
\end{aligned}
$$

Fig. 4. The Trickle Algorithm

time units. If a node hears an old metadata, it broadcasts the code necessary to update the node sending the old metadata. If it hears a new metadata (e.g., a larger version number), it broadcasts its own metadata, which triggers the receiver to send the updated program. To reduce the number of communication, each node announces only a limited number of times during each gossip. Through time, a network of thousands of sensor nodes shall be updated with the new code. The algorithm has been evaluated with extensive simulation in [21].

The modeling is presented in Figure 4. The main process of each sensor node (called *mote* in [20]) consists of three components running in parallel. Process $Routine$, which is reconfigurable, performs the routine task. Process $Update$ updates the node if an updated version of $Routine$ has been received. Process $Gossip$ models the "polite gossip" policy which is used to discover whether the node is outdated or not. In order to keep the mote up-to-date, a mote records a number of parameters, i.e., $version$ records the version of $Routine$, a counter, a threshold, a timer and a constant tau. Process $Routine$ periodically senses data from the external environment. Process $Update$ keeps waiting for an input on channel c (with message tag pro). Once such a gradient is received, both $version$ and $Routine$ are updated if the received code is newer. Process $Gossip$ captures the essence of the algorithm. The mote announces its current version on channel $meta$ once a while (specified by $Talk$) or it listens to other motes and reacts (specified $Reply$). In the process $Talk$, after t time units, the mote checks if it has talked too much ($counter \geq threshold$). If it has, the mote will keep silent until the next gossip. Otherwise, it will broadcast its version on channel $meta$ and wait for the next turn to talk. Every tau time units, the process restarts (and resets the counter). In the process $Reply$, when a mote hears a $version$ identical to its own ($x = version$), it increments $counter$. If the mote hears a version greater than its own ($x > version$), it broadcasts its own version, which will trigger a receiver to send the updated program. If the mote hears an old version, it broadcasts the code necessary to update the sending mote.

We believe that formal specification is a starting point for a number of formal analysis tasks. It is natural to ask whether this algorithm satisfies important safety and liveness properties. In this experiment, we reuse existing state-of-the-art verification support for real-time systems (e.g., UPPAAL [19]) to reason about the algorithm. In order to model check the algorithm, we must first close the system by explicitly specifying the environment. The following components are part of the environment, i.e., the network topology, the different versions of $Routine$ and the external data resource from which the sensor nodes collect data. As discussed above, the network topology may be specified using pre-defined variables (e.g., $location$ and $range$). There is a link between two nodes if and only if the nodes are within each other's range. In UPPAAL, however, processes communicate only through pre-defined channels. Thus, we need to pre-process the network topology and define channels for each link between the nodes, which is statically done given the values of the variables are not changing. One feature which is missing from timed automata is higher order processes. To the best of our knowledge, there are few tools which support verification of higher-order processes. Thus, higher-order processes are reduced to ordinary processes whenever the system is closed. This may not be always desirable or possible. For this experiment, because the details of $Routine$ are irrelevant, we simply abstract it away. In UPPAAL, timed automata are extended with broadcasting channels and committed states, which can be used to mimic broadcasting in ASP. Because the sensed data is irrelevant, it is simply ignored.

In our previous work, we have developed a systematic translation from Timed CSP to timed automata [10]. We defined a rich set of compositional operators for timed automata, which corresponds to the compositional operators in Timed CSP. By following the same approach, we develop a systematic translation from ASP to timed automata. Figure 5 presents the generated automata. Automaton (a) corresponds to the process $Update$. Because passing data through channels is not allowed in UPPAAL, we use shared variables to communicate instead, i.e., message sending writes a shared variable whereas message receiving reads the value. The state after the synchronization is marked as urgent so as to read the input value immediately. The parallel composition of automaton (b) and (c) corresponds to process $Gossip$. Adapting the timed patterns defined in [10], the timed interrupt $\triangledown_{tau}$ has been resolved using extra state invariants and transition guards. For instance, automaton (b) which loosely corresponds to process $Talk$ is modified to guarantee that whenever $x == tau$, the system is restored to the initial state. Automaton (c) loosely corresponds to process $Reply$.

Table 6 shows the experiment results using UPPAAL to verify instances of the algorithm. The network topologies are randomly generated with one constraint, i.e., all nodes are reachable. The results are obtains by executing UPPAAL 4.0.6 on Windows XP platform with Intel Core Duo 2.33GHz CPU and 3.25 GB memory. All models are deadlock-free as expected. A desired property of code propagation algorithms is that if a node is reachable, it will always eventually be updated. A counterexample is produced unexpectedly. Figure 7 elaborates the counterexample with a network containing 3 motes connected circularly. The link between nodes are directed as it is possible that node B hears A but node A cannot hear from B, e.g., A has a longer radio range. Initially, node A's version is 1, meaning that it is updated already. Once node A hears a meta-data from node C, it broadcasts its updated program. Only node B hears from

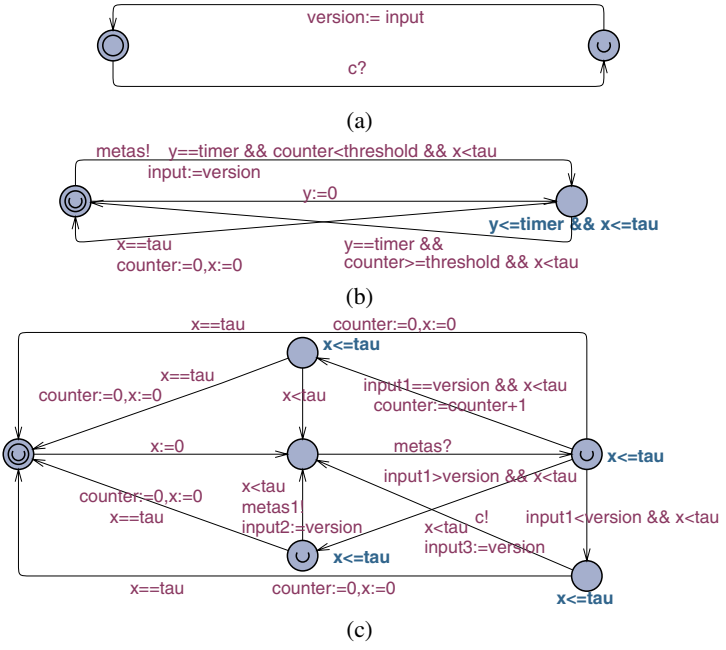


Fig. 5. UPPAAL Model of the Trickle Algorithm

Model	Property	time taken	Result
2 Motes	deadlock-free	< 1	true
3 Motes	same above	1	true
4 Motes	same above	12	true
5 Motes	same above	1080	true
6 Motes	same above	–	true
2 Motes	always-eventually all motes are updated	< 1	true
3 Motes	same above	1	false
4 Motes	same above	3	false
5 Motes	same above	25	false
6 Motes	same above	–	–

Fig. 6. Verifying Randomized Sensor Networks using UPPAAL

node A and thus node B is updated. Consequently, node C broadcasts its meta-data every time after receiving meta-data from node B and then node A broadcasts the updated program every time after receiving the meta-data from node C. However, because only node B receives from node A and it ignores the message since it has already been updated. As a result, node C is never updated. Notice that the property is true if all links are bi-directional or each node has fixed location and radio range. In order to prove the latter, we can show that if B hears A and A cannot hear from B, then the range of A

Fig. 7. A Counterexample

must be larger than that of B. Thus, a circular network like the one in Figure 7 is not possible. Nevertheless, such network topology is possible in practice.

For a network with 6 motes, UPPAAL needs significant amount time to verify the property. A typical sensor network application, however, may contain hundreds or thousands of sensor nodes. Observing that some distinguishable features are missing from UPPAAL (e.g., higher-order processes), this suggests that specialized verification mechanism and state space reduction techniques must be developed.

4.2 Face Routing Protocols

In this experiment, we review another example of sensor network application. We argue that current formal verification tools may not be sufficient to provide answers to natural questions on those systems.

Geographic routing algorithms are an important family of routing protocols of wireless ad hoc sensor networks. They have been shown to scale better than other alternatives, i.e., they require per node state that depends only on network density and not on network size. Established proposals include GFG [2], GPSR [17], etc. However, due to different forwarding mechanism, reachability between sensor nodes is not always guaranteed. In many of these algorithms (e.g., [17]), nodes forward packets to the neighbor closest to the destination whenever possible. The following process models the relevant behaviors of a node in the simplest greedy routing protocol.

$$
\begin{aligned}
Main = c?new(src, des, msg) \rightarrow \\
&([loc = des] \bullet message := msg;\ Skip\ \Box \\
&[src - des > loc - des] \bullet c!new(loc, des, msg) \rightarrow Main\ \Box \\
&[loc - des \geqslant src - des] \bullet Main)
\end{aligned}
$$

where loc is the location of the node and $message$ is a local data store recording the message received. In what follows, we assume that each node acquires its own position using GPS devices (as assumed in [17]). Without loss of generality, location is abstracted as a natural number. The message is composed of three parts, i.e., src is the location of the sender, des is the location of the intended receiver and msg is the message content itself. Each node tries to forward the message to its neighbors who are geographically closer to the destination than the node itself (i.e., $src - des > loc - des$). If the location of the sensor is the same as the destination, it means that message has reached the destination. If the sensor is not the destination and it is closer to the destination than the the sender, it would go on broadcasting the message with the sender location replaced by its own location. Otherwise it will discard this message. In [17], a simple beaconing algorithm provides all nodes with their neighbors' positions: periodically, each node broadcast a beacon which its own identifier and position. For simplicity,

we skip the modeling of the beaconing algorithm and assume that the messages are always broadcasted in the above modeling. Nonetheless, this modeling shares the same pitfall with the one in [17].

A critical requirement for any routing protocol is that, given a static network topology and reliable link layer support, the protocol must guarantee message delivery to a reachable node. It has been shown that message delivery is not always guaranteed for face routing algorithms [13]. For instance, there are topologies in which the only route to a destination requires a packet move temporally further in geometric distance from the destination, due to the presence of routing holes. Refer to [13] for more complicated protocols as well as pitfalls. Given a newly designed protocol, it is thus desirable to answer the question whether there exists a network topology such that the protocol does not function as expected. Different from traditional model checking, a model satisfying different constraints including temporal logic properties (e.g., always eventually the message is delivered) must be constructed and presented as a counterexample. This problem may be categorized as a model satisfiability problem. Because it concerns temporal logic constraints, bounded model checking techniques must be applied.

One solution is to formula the question as a Boolean formula and then apply state-of-the-art SAT-solvers to generate solutions automatically. In our previous work [29], we have developed ways of encoding compositional processes as SAT problems for bounded model checking. By applying a similar approach, our primary experiments show that for face routing protocols, not only we can prove/disprove desirable properties given a network topology but also generate one particular network topology which makes the given algorithm faulty. Given a bound on the number of nodes, and the data range of the pre-defined variable $location$ and $range$ (which identifies the network topology), a fixed number of Boolean variables are used to represent status of each node. The behaviors of each node can be translated to a labeled transition system (by applying the operational semantic) and then encoded as a Boolean formula in the standard way [9]. The encoded sensor nodes are composed using a similar approach proposed in [29]. After composing with the Boolean formula which represents the property (e.g., if there is a path from the source to the destination, then the message must always eventually reach the destination), an SAT-solver is used to assign *true/false* value to all Boolean variables. An assignment to the variables representing $location$ and $range$ identifies a network topology in which the protocol can not guarantee message delivery. In the above example, we have successfully generated connectivity graphs which contains routing holes. We are currently extending our tool [29] to fully automate the process. We have implemented and experimented a number of face routing protocols (like the above one and GFG). However, because of the size of sensor network applications, our prototype implementation must be extended with partial order reduction as well as symbolic techniques before practical usage.

5 Conclusion

In this paper, we proposed a high-level formal specification language specifically for wireless sensor networks. Unique language constructs have been defined to cope with the unique characteristics of such systems, e.g., sensing and actuating, inter-sensor

messaging, etc. Next, we developed a formal semantics for ASP. Lastly, we demonstrated how to use ASP to model sensor network applications as well as how to verify those models. From the examples, ASP showed its high expressiveness and conciseness in formally specifying the communications and behaviors of sensor network systems. In summary, ASP not only offers a way of modelling/specifying sensor networks, with the precise semantics defined, but also gives us a starting point for formal sensor network simulation, verification and synthesis.

We are currently developing a series of tools based on ASP. e.g., a high-level simulator, a verifier and a synthesizer (e.g., [28, 7, 11]). Existing simulators for sensor networks like TOSSIM and ns-2 mimic physical environment rather closely and thus provide rich simulation results. Nevertheless, they may not be as abstract as expected by system designers who are only interested in the high-level functionality of the system. For instance, a protocol designer would be firstly interested in if "in theory" the newly designed protocol is free of deadlocks and livelocks and then analysis the performance of the protocol with realistic settings. We are developing a simulator based on ASP for high-level simulation, based on the operational semantics presented in Section 3. A promising technique to handle large number of sensor nodes is the symbolic simulation proposed in [15]. One in-expensive approach to connect ASP to the current practise of sensor networks programming is by providing a transformation (which has been planned) from ASP to languages like $nesC$ which is designed to embody the execution model of TinyOS [14].

Acknowledgement

This work is partially supported by the research project "Sensor Networks Specification and Validation" (R-252-000-320-112) funded by Ministry of Education, Singapore.

References

1. Akyildiz, I.F., Su, W., Sankarasubramaniam, Y., Cayirci, E.: Wireless Sensor Networks: a Survey. Computer Networks 38(4), 393–422 (2002)
2. Bose, P., Morin, P., Stojmenovic, I., Urrutia, J.: Routing with Guaranteed Delivery in Ad Hoc Wireless Networks. Wireless Networks 7(6), 609–616 (2001)
3. Botts, M.: Sensor Model Language (SensorML) (2006), http://vast.nsstc.uah.edu/SensorML/
4. Boulis, A., Han, C.C., Srivastava, M.B.: Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In: Proceedings of International Conference on Mobile Systems, Applictions, and Services (MobiSys 2003) (2003)
5. Cardelli, L., Gordon, A.D.: Mobile Ambients. Theoretical Computer Science 240(1), 177–213 (2000)
6. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378. Springer, Heidelberg (1998)
7. Chen, C., Dong, J.S., Sun, J.: A verification system for timed interval calculus. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), pp. 271–280. ACM Press, New York (2008)

8. Ciancio, A.G., Pattem, S., Ortega, A., Krishnamachari, B.: Energy-efficient Data Representation and Routing for Wireless Sensor Networks based on a Distributed Wavelet Compression Algorithm. In: Proceedings of Information Processing in Sensor Networks (IPSN 2006), pp. 309–316 (2006)
9. Cimatti, A., Clarke, E.M., 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)
10. Dong, J.S., Hao, P., Qin, S.C., Sun, J., Wang, Y.: Timed Patterns: TCOZ to Timed Automata. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 483–498. Springer, Heidelberg (2004)
11. Dong, J.S., Hao, P., Sun, J., Zhang, X.: A Reasoning Method for Timed CSP Based on Constraint Solving. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 342–359. Springer, Heidelberg (2006)
12. Estrin, D., Govindan, R., Heidemann, J., Kumar, S.: Next Century Challenges: Scalable Coordination in Sensor Networks. In: Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom 1999), pp. 263–270 (1999)
13. Frey, H., Stojmenovic, I.: On Delivery Guarantees of Face and Combined Greedy-face Routing in Ad Hoc and Sensor Networks. In: Proceedings of the 12th Annual International Conference on Mobile Computing and Networking (MOBICOM 2006), pp. 390–401 (2006)
14. Gay, D., Levis, P., von Behren, J.R., Welsh, M., Brewer, E.A., Culler, D.E.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003 (PLDI 2003), pp. 1–11. ACM Press, New York (2003)
15. Goel, A., Meng, S., Roychoudhury, A., Thiagarajan, P.S.: Interacting process classes. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), pp. 302–311. ACM Press, New York (2006)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
17. Intanagonwiwat, C., Govindan, R., Estrin, D.: Directed Diffusion: a Scalable and Robust Communication Paradigm for Sensor Networks. In: Proceedings of International Conference on Mobile Computing and Networking (MOBICOM 2000), pp. 56–67 (2000)
18. 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)
19. Larsen, K.G., Pettersson, P., Wang, Y.: Uppaal in a Nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
20. Levis, P., Culler, D.E.: Maté: a Tiny Virtual Machine for Sensor Networks. In: Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002), pp. 85–95 (2002)
21. Levis, P., Patel, N., Culler, D.E., Shenker, S.: Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In: Proceedings of Networked Systems Design and Implementation (NSDI 2004), pp. 15–28 (2004)
22. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: an Acquisitional Query Processing System for Sensor Networks. ACM Transactions on Database Systems 30(1), 122–173 (2005)
23. Mahony, B., Dong, J.S.: Timed Communicating Object Z. IEEE Transactions on Software Engineering 26(2) (February 2000)
24. Mezzetti, N., Sangiorgi, D.: Towards a Calculus For Wireless Systems. Electronic Notes in Theoretical Computer Science 158, 331–353 (2006)
25. Prasad, K.V.S.: A Calculus of Broadcasting Systems. In: Abramsky, S. (ed.) TAPSOFT 1991. LNCS, vol. 493, pp. 338–358. Springer, Heidelberg (1991)

26. Schneider, S.: An Operational Semantics for Timed CSP. Information and Computation 116(2), 193–213 (1995)
27. Schneider, S., Davies, J., Jackson, D.M., Reed, G.M., Reed, J.N., Roscoe, A.W.: Timed CSP: Theory and practice. Real-Time: Theory in Practice 600, 640–675 (1992)
28. Sun, J., Dong, J.S.: Design Synthesis from Interaction and State-Based Specifications. IEEE Transactions on Software Engineering 32(6) (2006)
29. Sun, J., Liu, Y., Dong, J.S., Sun, J.: Bounded Model Checking of Compositional Processes. In: Proceedings of the Second IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 23–30. IEEE Computer Society Press, Los Alamitos (2008)
30. Thomsen, B.: Calculi for Higher Order Communicating Systems. PhD thesis (1990)
31. Tschirner, S., Xuedong, L., Yi, W.: Model-Based Validation of QoS Properties of Biomedical Sensor Networks. In: Proceedings of the International Conference on Embedded Software (EMSOFT 2008) (accepted, 2008)
32. von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active Messages: A Mechanism for Integrated Communication and Computation. In: Proceedings of International Symposium on Computer Architecture 1992 (ISCA 1992), pp. 256–266 (1992)

Appendix A: Basic Operational Semantics

$$\frac{}{(Stop, B) \overset{\tau}{\hookrightarrow} (Stop, B)} \qquad \frac{}{(Skip, B) \overset{\checkmark}{\hookrightarrow} (Stop, B)}$$

$$\frac{}{(e \rightarrow P, B) \overset{e}{\hookrightarrow} (P, B)} \qquad \frac{(P, B) \overset{\lambda}{\hookrightarrow} (P', B') \quad \lambda \neq \checkmark}{(P;\ Q, B) \overset{\lambda}{\hookrightarrow} (P';\ Q, B')}$$

$$\frac{(P, B) \overset{\checkmark}{\hookrightarrow} (P', B)}{(P;\ Q, B) \overset{\checkmark}{\hookrightarrow} (Q, B)} \qquad \frac{(P, B) \overset{\lambda}{\hookrightarrow} (P', B')}{(P \,\Box\, Q, B) \overset{\lambda}{\hookrightarrow} (P', B')}$$

$$\frac{(P, B) \overset{\lambda}{\hookrightarrow} (P', B')}{(P \,\triangledown\, Q, B) \overset{\lambda}{\hookrightarrow} (P' \,\triangledown\, Q, B')} \qquad \frac{(Q, B) \overset{\lambda}{\hookrightarrow} (Q', B')}{(P \,\triangledown\, Q, B) \overset{\lambda}{\hookrightarrow} (Q', B')}$$

$$\frac{(P, B) \overset{\lambda}{\hookrightarrow} (Skip, B)}{(P \,\triangledown\, Q, B) \overset{\lambda}{\hookrightarrow} (Skip, B)} \qquad \frac{\lambda \notin X \quad (P, B) \overset{\lambda}{\hookrightarrow} (P', B')}{(P \,|[\, X \,]|\, Q, B) \overset{\lambda}{\hookrightarrow} (P' \,|[\, X \,]|\, Q, B')}$$

$$\frac{e \in X \quad (P, B) \overset{e}{\hookrightarrow} (P', B) \quad (Q, B) \overset{e}{\hookrightarrow} (Q', B)}{(P \,|[\, X \,]|\, Q, B) \overset{e}{\hookrightarrow} (P' \parallel Q', B)}$$

Appendix B: Operational Rules for Inter-sensor Messaging

$$\frac{}{(c?X \to P, B) \overset{c?v}{\hookrightarrow} (P \oplus \{X \mapsto v\}, B)} \; [\; BI \;] \qquad \frac{}{(c?v \to P, B) \overset{c?v:}{\hookrightarrow}} \; [\; DisI \;]$$

$$\frac{}{(c!v \to P, B) \overset{c!v}{\hookrightarrow} (P, B)} \; [\; BO \;] \qquad \frac{}{(c!v \to P, B) \overset{c!v:}{\hookrightarrow}} \; [\; DisO \;]$$

$$\frac{(P_1, B_1) \overset{c!v}{\hookrightarrow} (P_1', B_1) \quad (P_2, B_2) \overset{c?v}{\hookrightarrow} (P_2', B_2')}{(P_1, B_1) \;|||\; (P_2, B_2) \overset{c!v}{\hookrightarrow} (P_1', B_1) \;|||\; (P_2', B_2')} \; [\; Broadcast \;]$$

$$\frac{(P_1, B_1) \overset{c?v}{\hookrightarrow} (P_1', B_1') \quad (P_2, B_2) \overset{c?v}{\hookrightarrow} (P_2', B_2')}{(P_1, B_1) \;|||\; (P_2, B_2) \overset{c?v}{\hookrightarrow} (P_1', B_1') \;|||\; (P_2', B_2')} \; [\; IParallel \;]$$

$$\frac{(P_1, B_1) \overset{\beta:}{\hookrightarrow} \quad (P_2, B_2) \overset{\beta:}{\hookrightarrow}}{(P_1, B_1) \;|||\; (P_2, B_2) \overset{\beta:}{\hookrightarrow} (P_1, B_1) \;|||\; (P_2, B_2)} \; [\; BJointDiscard \;]$$

$$\frac{(P_1, B_1) \overset{\beta:}{\hookrightarrow} \quad (P_2, B_2) \overset{\beta}{\hookrightarrow} (P_2', B_2')}{(P_1, B_1) \;|||\; (P_2, B_2) \overset{\beta}{\hookrightarrow} (P_1, B_1) \;|||\; (P_2', B_2')} \; [\; BDiscard(1) \;]$$

$$\frac{(P_1, B_1) \overset{\beta}{\hookrightarrow} (P_1', B_1') \quad (P_2, B_2) \overset{\beta:}{\hookrightarrow}}{(P_1, B_1) \;|||\; (P_2, B_2) \overset{\beta}{\hookrightarrow} (P_1', B_1') \;|||\; (P_2, B_2)} \; [\; BDiscard(2) \;]$$

Correct Channel Passing by Construction*

Chao Cai[1], Zongyan Qiu[1], Xiangpeng Zhao[1], and Hongli Yang[2]

[1] LMAM and Department of Informatics, School of Math.,
Peking University, Beijing 100871, China
{caic,qzy,zxp}@math.pku.edu.cn
[2] College of Computer Sciences, Beijing University of Technology,
Beijing 100022, China
yhl@bjut.edu.cn

Abstract. Channel passing is a mechanism to describe dynamic composition of parallel systems. As to Web services, both WS-BPEL and WS-CDL adopts this mechanism to support dynamic business processes. Web service composition may suffer from channel passing, e.g., some service might not get a correct channel to complete an interaction, and then the whole system would get stuck. The work presented here is aimed at designing services which are immune to channel problems. Firstly, we define a pair of model languages on both global and local levels with formally defined semantics. Based on these languages, we propose a top-down design methodology that generates local-level processes from a global specification. Finally, we give out a set of conditions for global specifications, from which the generated processes are guaranteed correct.

Keywords: Choreography, Orchestration, Channel passing, Web Services, Formal Methods.

1 Introduction

Web services promise the interoperability of various applications running on heterogeneous platforms over the Internet, which have gained increasing attention. Web service composition refers to the process of combining existing Web services to offer value-added services to support inter-organization or crossing-organization business processes.

Two levels of views to the composition of Web services exist, namely orchestration (local view) and choreography (global view). On one hand, an orchestration describes interactions from one service (peer) to the others from the view of this peer, for the implementation of a new service. The *de facto* standard for orchestration is WS-BPEL [2]. On the other hand, a choreography abstracts a category of peers as a role, and focuses on the composition of a group of roles from a global perspective. Choreography is used to specify business protocols from a global viewpoint, such that the protocol can be implemented by a group

* Supported by National Natural Science Foundation of China (No. 60603033 and No. 60773161).

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 338–354, 2008.

of individual distributed peers without a central point of control. WS-CDL [1] is a W3C candidate recommendation designed for describing service choreography.

Due to the heterogenous nature of Web service composition, it is important to find a way to describe how different service providers/consumers located in different places can be (possibly dynamically) connected with each other. The concept of channel is proposed for this purpose. Channels abstract the communication mechanism of Web services. A channel is seen as a virtual connection between two peers for communications. In other words, the only mean to pass messages in a Web service composition is to use a channel. There are different terminologies for channel in several languages.

In WS-BPEL, a peer communicates with other peers through *partner links*. A partner link is syntactically represented by an *endpoint reference*, which specifies where and how to invoke a service. Also, a peer exposes a WSDL [11] interface, where each endpoint is associated with a network address. In the following of this work, a partner link is abstracted as a channel variable, while an endpoint reference as a channel. In WS-CDL, the concept of *channel variable* is introduced. A channel variable captures information of a channel.

In an ideal situation, all peers of a business process know the channels that they will use before the business process starts. When all channels are known by their users and all the usage of the channels are described statically in the business process specification, we say that the process has a static communication structure. Most formal work on service composition adopted this assumption.

However, in the real world applications, the static communication structure may not be sufficient. In a multi-party business process that involves several peers, some of these peers may be selected dynamically during the execution by other peers who have already joined the work. Also, a peer may not take part in a process until some specific event happens in the execution. If a peer needs to join the interaction dynamically during the execution and then communicate and cooperate with the other peers, it has to wait until it obtains some necessary channels from the peers which are already in the work.

For example, in an online shopping process involving a buyer, a seller and a shipper, the shipper will not join the process until the deal is agreed by the buyer and the seller. To complete the business process, the shipper should get the channel information from, for example, the seller, and then use the information to contact the buyer. Of course, the channel information must be transferred through a channel too. The term *channel passing* means that channel is passed through a message exchange. Channel passing is critical in supporting complex business processes.

Both WS-BPEL and WS-CDL include some mechanisms to support the specification of processes with dynamic communication structures and channel passing. In WS-BPEL, two ways to initialize a channel variable are permitted. On one hand, the channel variable can be bound with a channel statically, either via an assignment in the business process definition, or as part of the process deployment. On the other hand, it is also possible to bind channel variables dynamically. The service could receive a channel from some partner and assign it

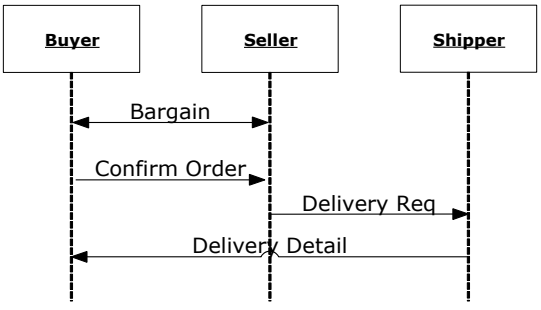


Fig. 1. Online shopping

to a channel variable at runtime. In WS-CDL, channel passing is syntactically supported by interactions. Channels can be transferred through interactions, and then the transferred channels can be used in subsequent interactions.

Figure 1 shows an elaborated description of the online shopping example discussed above in the form of UML sequence diagram. When the buyer and the seller agree on the price of the goods, a shipper will be designated to transport the goods from the seller's warehouse to the buyer. In this scenario, the shipper is dynamically selected by the seller according to some factors, e.g., the location of the buyer, or the quality of goods. Clearly, the chosen shipper has no idea where the buyer is unless the seller tells her/him the buyer's contact information, which can be either a phone number or an address in an off-line transaction. We will meet similar situation if the shopper is selected by the buyer.

In the Web service environment, the contact information must contain the information of the contacting channels. Thus the "channel passing" is indispensable. When the channels need to be passed to fulfill the communication requirement, and the communication structures are dynamically organized, the business process designers will face new challenges to avoid:

Channel-Absent. They should ensure that each peer can get necessary channels for its communication. In the example, we need guarantee that the shipper is able to obtain a channel for its work on the delivery.

Channel-Mistake. They should ensure that each peer uses the correct channel in every communication. In the example, we must guarantee that the shipper use the correct channel to contact the desired buyer.

From the theoretical viewpoint, the first challenge is a liveness requirement, and the second a safety requirement. The designers will be very glad if we propose some method for them to check the design problems of this kind, or develop some methodology to completely avoid these defects.

The WS Choreography Model Overview [5] requires that a choreography can be used to generate individual behavioral interface of peers. The top-down approach has been proposed and proved effective in some theoretical study [13, 10, 18], in the sense that the generated individual peers are behaviorally equivalent with the

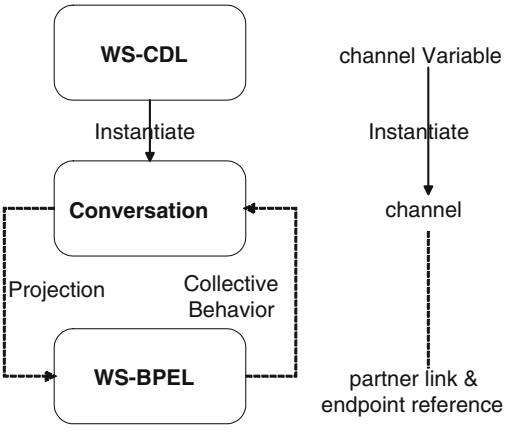


Fig. 2. Conversation

global specification on some conditions. When we think about the dynamic composition structures, with the consideration of channel passing, this problem must also be revisited.

To face these challenges, in this paper, we define a pair of formal languages with channel passing features for the global and local Web service specifications respectively. We call them a conversation language and a peer language. Based on these models, we show how to reason about channel-related problems on both levels. In addition, we propose a projection from a global specification written in the conversation language to a set of implementations in the peer language. To resolve the challenges mentioned above, some conditions for global specifications are proposed, and we prove that the peers generated from a conversation satisfying out conditions can be composed to a system, which will never suffer from channel-absent and channel-mistake defects. In this study, we take conversation as a simplified model of choreography. The difference between conversations and choreographies will also be discussed.

The rest of the paper is organized as follows. A model of conversation is given in Section 2. Section 3 presents a simple orchestration language with channel passing, and defines its operational semantics. A projection from conversation to peers and its correctness is given in Section 4. Section 5 gives a case study to demonstrate the projection and the correctness constraints. Some related work and discussion are presented in Section 6. In Section 7, we conclude the work.

2 A Conversation Language

We present a language *CONV* for global specification in this section. A document written in *CONV* is called a *conversation*.

Conversations specified in *CONV* are different from choreographies in WS-CDL in three aspects. Firstly, a conversation specifies a group of peers and their

collaborative behavior. In contrast, choreography describes interactions among a group of roles, while a role is an abstraction of some peers which have similar behavior and functionality. In other words, a peer can be seen as an instance of a role, and a conversation is thus an instance of a choreography, as depicted in Fig. 2. Secondly, it is channel instance through which the peers communicate with each other in conversation, while channel variables are used in choreography.

The third point is a structural addition, because here we include the description of the initial channel set for each peer, which has no counterpart in WS-CDL. The initial channel sets are important, and we discussed this problem in another work [8]. Briefly, without the specification of initial channel sets for the peers, we can not have automatic method to check the design defects related to channels, similar to if we did not have type definitions, it would be not possible to have type-checking.

2.1 Syntax

In a conversation, each peer has an indexed name that ranges over r, r^1,..., r^n. Peers communicate with each other through channel instances (or simply, channels), that range over c, c_1, $c_2, \cdots$.

A conversation consists of a group of peer declarations, and also an activity which specifying the collaborative behavior of the peers, with the form:

$$conv \ ::= [(\overline{PDecl}), A]$$

A peer declaration has the form of:

$$PDecl \ ::= \ r[C_{\mathsf{k}}, C_{\mathsf{in}}]$$

Here r is the name of the peer. The declaration specifies also an initial channel set C_{k} known by the peer before the conversation begins, and a set of channels C_{in} that the peer listens on. A peer can only receive messages through the channels on which it listens.

Each peer involved in collaboration will execute a series of activities. The basic activities can be divided into two kinds: the local activities which are performed by a single peer, and the communication activities from one peer to another. We will use meta-variables $a^i, a^i_1, ...$ for local activities performed by peer r^i. A communication from r^i to r^j through channel c_1 is represented as $c_1 : r^i \xrightarrow{c_2} r^j$. Here c_2 is the passed channel in the activity which can be used by receiver r^j in the communication afterward. Multiple channels are permitted to be transferred in one interaction in WS-CDL, which can be simulated by several interactions in our model, which transfer one channel each time. Since here we care only about the channel passing, a communication without channel passing will be represented as $c_1 : r^i \longrightarrow r^j$.

The control structures presented in the language are very simple. Sequential and parallel composition structures are employed here and the behavior of a conversation is simply an activity A, with the syntax as:

$$
\begin{array}{lll}
A ::= & c_1 : r^i \xrightarrow{c_2} r^j & \text{(interaction)} \\
| & a^i & \text{(local activity)} \\
| & A_1 ; A_2 & \text{(sequence)} \\
| & A_1 \mid A_2 & \text{(parallel)}
\end{array}
$$

2.2 Semantics

In this subsection, we present the operational semantics of the language *CONV*.

Firstly, we need to introduce some auxiliary notations for the semantic definitions. The binary predicate $\mathcal{K}(r, c)$ means that peer r knows channel c. When $\mathcal{K}(r, c)$ holds, of course, r can use c in the communications. We promote $\mathcal{K}$ to the set of channels. If $C = \{c_1, .., c_k\}$ is a set of channels, then $\mathcal{K}(r, C) \widehat{=} \mathcal{K}(r, c_1) \wedge \ldots \wedge \mathcal{K}(r, c_k)$. We use *pro*, possibly with subscription or primes, to denote a proposition made from $\mathcal{K}$, false and $\wedge$, which is used to represent an *initial channel set* of some peers and *state* in the semantics. It is worth to note that, the state here includes only information about roles and the channel sets they know. This reflects our main focus in this work.

We represent the semantics by transition rules between configurations. A configuration is of the form (pro, A), where A is an activity, *pro* is state. For a conversation $[(\overline{r^i[C^i_{\mathsf{k}}, C^i_{\mathsf{in}}]}), A]$, its initial state is $pro_0 = \bigwedge_i \mathcal{K}(r^i, C^i_{\mathsf{k}})$, that is, before the conversation start, each role r^i knows its set C^i_{k}.

A local activity of any peer can always execute. It does not change the state.

$$(pro, a^i) \longrightarrow (pro, \epsilon) \qquad \text{(Local Activity)}$$

A communication from r^i through c_1 can happen only if the sender r^i knows the channel c_1. Besides, if a channel is passed in the communication, the sender must know that channel as well. After the interaction, the receiver will know the passed channel too, and can use it in the communication consequently.

$$\frac{pro \vdash \mathcal{K}(r^i, \{c_1, c_2\})}{(pro, c_1 : r^i \xrightarrow{c_2} r^j) \longrightarrow (pro \wedge \mathcal{K}(r^j, c_2), \epsilon)} \qquad \text{(Communication)}$$

We need another rule for the communication without channel passing. It is almost the same as this rule, and is omitted here.

Parallel composition is easily handled here. All parallel branches run independently, and then the resulting states are combined.

$$\frac{(pro, A_1) \longrightarrow (pro_1, \epsilon) \quad (pro, A_2) \longrightarrow (pro_2, \epsilon)}{(pro, A_1 \mid A_2) \longrightarrow (pro_1 \wedge pro_2, \epsilon)} \qquad \text{(Parallel)}$$

This rule reflects the most restrictive conditions, under which each branch can finish its interactions independently and the branches can be executed in any order.

The transition rule for sequential compositions is regular.

$$\frac{(pro, A_1) \longrightarrow (pro', \epsilon) \qquad (pro', A_2) \longrightarrow (pro'', \epsilon)}{(pro, A_1 ; A_2) \longrightarrow (pro'', \epsilon)} \qquad \text{(Sequence)}$$

Based on the operational semantics, we develop an algorithm to check whether a conversation lacks of channels which is described by predicate ϕ. If activity A gets stuck in state pro because of lack of channel, $\phi(A, pro)$ evaluates to false.

$$\begin{aligned}
&\phi(a^i, pro) = pro \\
&\phi(c_1 : r^i \xrightarrow{c_2} r^j, pro) = \begin{cases} \mathsf{false} & \text{if } pro \not\vdash \mathcal{K}(r^i, \{c_1, c_2\}) \\ pro \wedge \mathcal{K}(r^j, c_2) & \text{else} \end{cases} \\
&\phi(A_1 \mid A_2, pro) = \phi(A_1, pro) \wedge \phi(A_2, pro) \\
&\phi(A_1; A_2, pro) = \phi(A_2, \phi(A_1, pro))
\end{aligned}$$

Theorem 1 (No Stuck). *For a conversation with activity A and initial state pro_0, if $\phi(A, pro_0)$ holds, then the execution of the conversation will not meet a state where a communication can not carry on because of lacking of channels.*

The proof of this theorem is simple and is omitted here.

3 A Peer Language

A conversation describes the interactions among peers from a global viewpoint. It is intended to be implemented by the coordination of a set of independent peers. In order to study the relationship between the globally described conversation and the behavior of the collaborative local activities, we define a language *PEER* for specifying the peers, and present its formal semantics.

The language is a simple process language with channel passing features. One notable point of the language is that we have two distinguishable levels: the peer (a process) and the system (a combination of several peers). Semantically, we do not have inter-peer communication and synchronization (i.e., we do not care about these in the study). The only communications here occur between different peers, which represent the collaborations between peers.

3.1 Syntax

The syntax of *PEER* is given below to specify individual services. Here a peer is an activity A. Peers are similar with activities in *CONV*, the only difference is communication activity. Concretely speaking, where the interaction in *CONV* is replaced by the sending and the receiving activities, and channels may be replaced by channel variables.

Here we use c to denote channel, x for channel variables, and h for either a channel or a channel variable. Variables with the same name have no relation with each other if they appear in different peers.

$$\begin{array}{rcll}
A & ::= & BA & \text{(basics)} \\
 & \mid & A;A & \text{(sequence)} \\
 & \mid & A \mid A & \text{(parallel)} \\
BA & ::= & \mathsf{skip} & \text{(silent act.)} \\
 & \mid & a & \text{(local act.)} \\
 & \mid & h_1!h_2 & \text{(send)} \\
 & \mid & c?x & \text{(receive)}
\end{array}$$

From the syntax, we know that a sending activity can use either channel variable or channel instance, and send a channel instance or the value of a channel variable. On the other hand, a receiving activity can only take message from a channel instance, and record the received channel to a variable. As usually, a sending activity can succeed only when some other peer is ready to receive it on the same channel. For our focus on channel passing, we omit messages except channels. Then the sending and receiving activities are $h!$ and $c?$ respectively, in the case what passed is not a channel.

Peers collaborate and communicate with each other through network. A system composed by a set of peers is defined as "$A_1 \parallel ... \parallel A_n$".

3.2 Semantics

We propose the operational semantics of *PEER*. Firstly we define the configuration of an individual peer, then the configuration of a system which involves several peers. The semantic rules are transitions between both configurations of peers and configurations of systems, which are given finally.

A peer is represent as a pair (σ, A) at run-time, where A is an activity to execute, and σ is a mapping from channel variables to channel instances. Initially, the configuration for a peer process A is $(\emptyset, A)$ where $\emptyset$ represents an empty map. We use $\sigma(x) = \perp$ to mean that x has not been assigned. For convenience, we define $\sigma(c) = c$ for each channel instance c. We use $\sigma[x \mapsto c]$ to denote a function override, which is a map similar to σ except that maps x to c.

A configuration of a system is represented as $(\sigma_1, A_n) \parallel ... \parallel (\sigma_n, A_n)$, which will be denoted as Σ, possibly with a subscription. An initial configuration of a system is composed by the initial configurations of each peer.

In each rule below, α denotes an action, which falls into four categories: local action a, sending action $c!c'$, receiving action $c?c'$ or silent action τ. We use $\mathsf{bn}(\alpha)$ to denote the bounded channel of α defined as

$$\begin{aligned} \mathsf{bn}(a) &= \emptyset \\ \mathsf{bn}(c!c') &= \{c\} \\ \mathsf{bn}(c?c') &= \{c\} \\ \mathsf{bn}(\tau) &= \emptyset \end{aligned}$$

Now, we give the transition rules. Firstly, a local activity or a silent activity of any peer can always happen without any influence of the state.

$$(\sigma, a) \xrightarrow{a} (\sigma, \epsilon) \qquad \text{(Local)}$$

$$(\sigma, \mathsf{skip}) \xrightarrow{\tau} (\sigma, \epsilon) \qquad \text{(Silent)}$$

The rules for sequential and parallel composition are regular.

$$\frac{(\sigma, A_1) \xrightarrow{\alpha} (\sigma', A_1')}{(\sigma, A_1; A_2) \xrightarrow{\alpha} (\sigma', A_1'; A_2)} \qquad \text{(Sequence)}$$

$$\frac{(\sigma, A_1) \xrightarrow{\alpha} (\sigma', A_1')}{(\sigma, A_1 \mid A_2) \xrightarrow{\alpha} (\sigma', A_1' \mid A_2)} \qquad \text{(Parallel)}$$

A sending activity through channel variable x_1 can occur only if the sender knows the corresponding channel instance of x_1, and that of the passed channel if there is channel passing.

$$\frac{\sigma(x_1) = c_1 \neq \bot, \quad \sigma(x_2) = c_2 \neq \bot}{(\sigma, x_1!x_2) \xrightarrow{c_1!c_2} (\sigma, \epsilon)} \qquad \text{(Send)}$$

If a peer listens on channel c_1, it may proceed one step, and the environment is updated by the receiving action.

$$(\sigma, c_1?x_2) \xrightarrow{c_1?c_2} (\sigma[x_2 \mapsto c_2], \epsilon) \qquad \text{(Receive)}$$

Communication happens if a sending activity and a receiving one match.

$$\frac{(\sigma_1, A_1) \xrightarrow{c_1!c_2} (\sigma_1', A_1'), \quad (\sigma_2, A_2) \xrightarrow{c_1?c_2} (\sigma_2', A_2')}{(\sigma_1, A_1) \parallel (\sigma_2, A_2) \xrightarrow{\tau} (\sigma_1', A_1') \parallel (\sigma_2', A_2')} \qquad \text{(Communication)}$$

From the rules defined, we see that the communication can happen only between different peers in a system, when one of them sends and another receives.

If a system (composed of some peers) can complete an action, then a bigger system (composed with some more peers) can also do this action.

$$\frac{\Sigma_1 \xrightarrow{\alpha} \Sigma_2}{\Sigma_1 \parallel \Sigma \xrightarrow{\alpha} \Sigma_2 \parallel \Sigma} \qquad \text{(Promotion)}$$

An equivalent rule and some congruence rules are given to reduce the number of transition rules.

$$\frac{\Sigma_0 \equiv \Sigma_1, \Sigma_1 \xrightarrow{\alpha} \Sigma_2, \Sigma_2 \equiv \Sigma_3}{\Sigma_0 \xrightarrow{\alpha} \Sigma_3} \qquad \text{(Equivalence)}$$

Here are the *congruence rules*:

$$\begin{gathered} A_1 \mid A_2 \equiv A_2 \mid A_1 \\ \epsilon; A \equiv A \\ \epsilon \mid A \equiv A \\ (\sigma_1, A_1) \parallel (\sigma_2, A_2) \equiv (\sigma_2, A_2) \parallel (\sigma_1, A_1) \end{gathered}$$

The restriction rule prohibits peer from communication with those outside the system through the channels listed in L.

$$\frac{\Sigma \xrightarrow{\alpha} \Sigma'}{\Sigma \setminus L \xrightarrow{\alpha} \Sigma' \setminus L} \qquad \text{if } \mathsf{bn}(\alpha) \cap L = \emptyset \qquad \text{(Restriction)}$$

Although Internet is very big and deep, a peer uses only a part of it to communicate with its partners, and the rest of the network can be excluded. The restriction rule reflects the fact. By this rule, we can restrict some communication in the boundary of a system.

Definition 1. *A system, in which each peer has finished their task, formally* $(\sigma_1, \epsilon) \parallel ... \parallel (\sigma_n, \epsilon)$*, is denoted as* 0*.*

Definition 2. *Given a system* $A_1 \parallel \cdots \parallel A_n$*, where its initial configuration is* Σ*, and* L *is the set of all channel instance involved, we say the system is* deadlock free*, if* $\Sigma \setminus L$ *never deadlocks. i.e., for any configuration* Σ'*, if* $\Sigma \setminus L \longrightarrow^* \Sigma' \setminus L$*, there is a reduction* $\Sigma' \setminus L \longrightarrow^* 0 \setminus L$*. Here* $\longrightarrow^*$ *is the transitive relation of* $\longrightarrow$*.*

We will propose a way to help designers getting deadlock free systems in the next section.

4 A Top-Down Design Approach

Conversations can be used to support the development of individual peers. Here we propose an algorithm to generate the implementations of the peers from a conversation. Then, we will prove that the generated systems can be guaranteed correctness if the original conversations satisfy a set of conditions.

4.1 Projection

To generate implementations, we need some auxiliary tools.

Firstly, for a given conversation, we take an arbitrary mapping θ_0, which maps channel instances appeared in the conversation to channel variables. e.g., $\theta_0(c) = x$, for each c in the conversation and some x. The only requirement for θ_0 is that it should be injective, that is, it gives different variable names for different channel instances.

With mapping θ_0 at hand, we define a function θ, which takes as input a channel instance and a peer's index, return either a channel instance or a channel variable. Its definition is as follows:

$$\theta(c, i) = \begin{cases} c & \text{if } c \in C^i_{\mathsf{k}} \cup C^i_{\mathsf{in}} \\ \theta_0(c) & \text{otherwise} \end{cases}$$

Intuitively, if $\theta(c, i)$ returns some variable x, we plan to use x as the variable to record channel c in role r^i when c is passed to this role. Clearly, if we use θ for different roles with the same channel, the returned names will be the same. However, it does not makes any conflicts, because variables reside on different roles, even with the same name, have nothing to do with each other.

The project algorithm π is given below. For a conversation written in *CONV* with behavior A, and the destination peer is r^k, $\pi(A, k)$ generates the implementation of r^k in *PEER*.

$$\pi(a^i, k) \mathrel{\widehat{=}} \begin{cases} a^i & \text{if } k = i \\ \mathsf{skip} & \text{if } k \neq i \end{cases}$$

$$\pi(c_1 : i \xrightarrow{c_2} j, k) \mathrel{\widehat{=}} \begin{cases} \theta(c_1, k)!\theta(c_2, k) & \text{if } k = i \\ \theta(c_1, k)?\theta(c_2, k) & \text{if } k = j \\ \mathsf{skip} & \text{otherwise} \end{cases}$$

$$\pi(A_1; A_2, k) \mathrel{\widehat{=}} \pi(A_1, k); \pi(A_2, k)$$

$$\pi(A_1 \mid A_2, k) \mathrel{\widehat{=}} \pi(A_1, k) \mid \pi(A_2, k)$$

It is not hard to see that the function θ defined before helps to keep the consistent using of each channel for generating a reasonable implementation.

If a conversation has n roles, we will get n peers by using this projection π. The combination of these peers forms a system. The relationship between the conversation and this system will be discussed in next subsection.

4.2 Relation between Conversation and Generated Peers

For a conversation which describes the correlative behavior of n peers, the projection π defined above will generate n individual peers (processes). We hope that the generated peers, when combined together to form a system, will be semantically equivalent to the conversation. Previous results show that such equivalence can only hold under some conditions on the global-viewed specifications [18,13,10]. We need some additional conditions for the situation here, as we have taken the channel passing into our consideration.

Now we propose a group of conditions for conversations that ensure correct channels and channel passing for the peers generated from it. These new conditions are orthogonal to the other conditions, so we do not repeat conditions and properties presented in previous works.

Theorem 2. *For a conversation* $[(\overline{r^i[C^i_{\mathsf{k}}, C^i_{\mathsf{in}}]}), A]$*, if:*

1. *For any two peers* r^i *and* r^j*,* $C^i_{\mathsf{in}} \cap C^j_{\mathsf{in}} = \emptyset$*, if* $i \neq j$*.*
2. *For each peer* r^j *and interaction* $c_1 : r^i \xrightarrow{c_2} r^j$*, we must have* $c_1 \in C^j_{\mathsf{in}}$*.*
3. $\phi(A, \bigwedge \mathcal{K}(r^i, C^i_{\mathsf{k}}))$ *does not evaluate to* false*.*

Then, the peers generated by π *combined together, will never suffer "Channel-Absent" or " Channel-Mistake".*

In the theorem, the first condition requires different peers to listen on different channels, thus ensures deterministic information flow. The second condition says that the receiver should declare all the channels it listens on. The third condition rules out these conversations which are lack of channels.

Theorem 2 can be divided into the following two sub-theorems:

Theorem 3 (Correct Channels). *If a conversation satisfies all the three conditions given in* **Theorem 2***, then each peer generated from it by projection* π *will get the desired channels. That is, for any interaction* $c_1 : r^i \xrightarrow{c_2} r^j$ *in the conversation, in the system composed by the generated peers, peer* r^j *will receive* c_2 *in the corresponding interaction.*

Theorem 4 (Sufficient Channels). *Suppose Conv is a conversation satisfies all the three conditions given in* **Theorem 2***, then the system composed of these generated peers is deadlock-free.*

These theorems solve the challenges of "Channel-Absent" and "Channel-Mistake". The proof for the theorems is given in Appendix A.

We must point out that only these three conditions can not ensure the behavioral equivalence between a conversation and the system formed by its generated

peers. An example is a conversation with activity $c_1 : r^1 \longrightarrow r^2; c_2 : r^3 \longrightarrow r^4$, where two interactions may be executed in any order in the generated system. The problem like this is a general control flow problem, but not specially related to channel passing, thus is out of range of this paper. Interested readers can refer our previous work [18], where we have a detailed study on this.

5 Case Study

Now we reconsider the example depicted in Figure 1, and write down that scenario formally in our language *CONV*. There are three peers in the conversation: a buyer "b", a seller "s", and a shipper "d" which takes charge of the goods delivery. These peers communicate through several channels. The buyer listens on c_b to receive messages, while c_s is the channel for the seller and c_d for the shipper.

Firstly, we give a defective conversation $Conv_1$ to see what will happen. For easy the discussion, we add a label for each interaction.

$$\begin{array}{rl} Conv_1 = [\ & (b[\{c_s, c_b\}, \{c_b\}], \\ & s[\{c_s, c_d\}, \{c_s\}], \\ & d[\{c_d\}, \{c_d\}]), \\ & I_1 : \quad c_s : b \longrightarrow s; \\ & I_2 : \quad c_d : s \longrightarrow d; \\ & I_3 : \quad c_b : d \longrightarrow b \\ ] & \end{array}$$

Using the projection π defined in Section 4, we will get the following three peer implementations b_1, s_1 and d_1:

$$\begin{array}{rcl} b_1 & = & c_s!; c_b? \\ s_1 & = & c_s?; c_d! \\ d_1 & = & c_d?; x! \end{array}$$

By the checking algorithm ϕ given in Section 2, we know that the conversation lacks of channel. Interaction I_3 can not be carried out because the shipper does not know the channel of the buyer. Also, by the reduction rules in Section 3, the system composed of b_1, s_1 and d_1 will eventually deadlock, because the shipper can not evaluate x to a channel instance.

Many methods can be applied to remedy this defect. A simple one is to let the shipper know c_b initially, as in the $Conv_2$ given here:

$$\begin{array}{rl} Conv_2 = [\ & (b[\{c_s, c_b\}, \{c_b\}], \\ & s[\{c_s, c_d\}, \{c_s\}], \\ & d[\{c_b, c_d\}, \{c_d\}]), \\ & I_1 : \quad c_s : b \longrightarrow s; \\ & I_2 : \quad c_d : s \longrightarrow d; \\ & I_3 : \quad c_b : d \longrightarrow b \\ ] & \end{array}$$

The peers generated from $Conv_2$ are as follows:

$$\begin{aligned} b_2 &= c_s!; c_b? \\ s_2 &= c_s?; c_d! \\ d_2 &= c_d?; c_b! \end{aligned}$$

Now, we can verify that both $Conv_2$ and "$(b_2 \parallel s_2 \parallel d_s) \setminus L$" are deadlock-free, where $(L = \{c_s, c_b, c_d\})$.

However, conversation $Conv_2$ is practically useful only if a static communication structure and the relationship between the three peers is predefined. Generally, as discussed in the Introduction, it is more reasonable that the seller selects shipper and tells it the channel of the buyer. For model this process, we can add channel passing from the buyer to the seller and then from the seller to the shipper, and finally have a conversation below:

$$\begin{aligned} Conv_3 = [\ & (b[\{c_s, c_b\}, \{c_b\}], \\ & s[\{c_s, c_d\}, \{c_s\}], \\ & d[\{c_d\}, \{c_d\}]), \\ & I_1 : \quad c_s : b \xrightarrow{c_b} s; \\ & I_2 : \quad c_d : s \xrightarrow{c_b} d; \\ & I_3 : \quad c_b : d \longrightarrow b \\ ] & \end{aligned}$$

The peers generated from $Conv_3$ are as follows:

$$\begin{aligned} b_3 &= c_s! c_b; c_b? \\ s_3 &= c_s? x; c_d! x \\ d_3 &= c_d? x; x! \end{aligned}$$

We can easily check that ϕ holds for the conversation, and the generated system behaves as desired.

Both revised versions work in the sense that channels are sufficient for all peers to complete their processes, and each group of peers generated from them are deadlock-free. But considering the actual business practice, it is not realistic that a shipper knows the customer's contact information before a deal begins.

The example gives us a hint that whether a peer can know/should know a partner's channel is application-dependent, and should be decided by the system designer. If a conversation is channel-defective, there are, in general, always more than one way to repair the defect. It is crucial in the practice to choose a right repairing method.

6 Related Work

Due to the growing interesting on Web services, some formal models on both service choreography and orchestration have been proposed, and realization and conformance problems are discussed based on the models.

In [6], Busi *et al.* formalized conformance with a bisimulation-like relation. In their another paper [7], the notion of state variables in the semantics of the orchestration model is used to build a relation between choreography to orchestration operationally. By means of automata, Schifanella *et al.* [3] defined a conformance notion that tests whether interoperability is guaranteed. Fu *et al.* [14] specified a conversation protocol by a realizable Büchi automaton, and the peer implementations are synthesized from the protocol via projection. Bravetti and Zavattaro [4] proposed a theory of contracts for conformance checking. They defined an effective procedure that can be used to verify whether a service with a given contract can correctly play a specific role within a choreography. Moreover, Decker *et al.* discussed the issue of local enforceability of *Let's Dance* choreographies in [12]. Aalst [19] focuses on conformance by comparing the observed behavior recorded in logs with some predefined model. In a previous work [18], we defined the concept of *restricted natural choreography* that is easily implementable, and proposed two structural conditions as a criterion to distinguish the restricted natural choreography. However, these literatures did not take channel passing and dynamic communication structures into to their model.

Carbone *et al.* [9] studied the description of communication behaviors from both global message flows and end-point behavior levels respectively. They introduced the concept of *service channel* and *session channel*. More recently, Lucchi [17] adopted π-calculus to represent the semantics of WS-BPEL, and formalized channel passing as location mobility in paper [15]. To model Web services, Laneve and Zavattaro extended π-calculus to Webπ [16], which can describe both a single service (machine) and services compositions. These works put channel passing (or name passing) into models but did not consider the reasoning about channels, nor verification of channel sufficiency and correctness.

In [20], we proposed a small language for modeling the channel passing aspect of choreography. Based on the formal semantics of the language, some algorithms for static checking choreography and generating necessary channel passing are studied. We also show how to use this model to check design defects in WS-CDL where no initial channel sets for each participants are given in [8]. However, in these work, we have not considered the implementation problem of choreography.

7 Conclusion and Future Work

A channel contains information on where and how to send messages to a specific peer, which arises in diverse forms in practice, such as sockets, remote object IDs, and URLs [10]. Channels can be passed within message content for the receiver to use in the consequent communications. The so-called channel passing allows the destination for messages to be determined dynamically. Both WS-BPEL and WS-CDL adopt some mechanisms to support channel passing. When channel passing is involved, designers must ensure that each service can get enough and correct channels, i.e., getting rid of "Channel-Absent" and "Channel-Mistake" problems present in their system.

To solve these challenges, we propose a formal language *CONV* for the specify conversation and a top-down approach to generate individual peers from a conversation. Also, we define a formal model for peers to reason about channels. Based on these formalization, we propose three conditions for conversations, and prove that these condition are sufficient to restrict conversations so that generated peers are immune to the these problems.

To design a conversation, all related channels should be determined. But sometimes in practice, channels are generated at run-time to include correlation information. In this case, channel variables are used in global description. Then, the peers take charge of instantiation of some channels and different peers may give contradict values for a same channel variables. In future work, we will explore such global description, and study its realizability problem.

References

1. Web Services Choreography Description Language (WS-CDL), version 1.0 (2005), http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/
2. Business Process Execution Language for Web Services (WS-BPEL), version 2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
3. Baldoni, M., Baroglio, C., Martelli, A., Patti, V., Schifanella, C.: Verifying the conformance of web services to global interaction protocols: A first step. In: Bravetti, M., Kloul, L., Zavattaro, G. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 257–271. Springer, Heidelberg (2005)
4. Bravetti, M., Zavattaro, G.: Towards a unifying theory for choreography conformance and contract compliance. In: Proc. of Software Composition 2007. Springer, Heidelberg (2007)
5. Burdett, D., Kavantzas, N.: WS Choreography Model Overview, http://www.w3.org/TR/ws-chor-model/
6. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration: A synergic approach for system design. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 228–240. Springer, Heidelberg (2005)
7. Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)
8. Cai, C., Qiu, Z.: An approach to check choreography with channel passing in WS-CDL. In: The Proceeding of International Conference on Web Service (ICWS) (to appear, 2008)
9. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centred programming for web services. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 2–17. Springer, Heidelberg (2007)
10. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A theoretical basis of communication-centred concurrent programming. Technical report, W3C (2006), http://www.w3.org/2002/ws/chor/edcopies/theory/note.pdf
11. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Service Definition Language (WSDL) 1.1, http://www.w3.org/TR/wsdl

12. Decker, G., Weske, M.: Local enforceability in interaction petri nets. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 305–319. Springer, Heidelberg (2007)
13. Fu, X., Bultan, T., Su, J.: A top-down approach to modeling global behaviors of web services. In: REOS 2003 (2003)
14. Fu, X., Bultan, T., Su, J.: Conversation protocols: A formalism for specification and verification of reactive electronic services. Theoretical Computer Science 328 (2004)
15. Guidi, C., Lucchi, R.: Mobility mechanisms in service oriented computing. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 233–250. Springer, Heidelberg (2006)
16. Laneve, Zavattaro: Foundations of web transactions. In: Myaeng, S.-H., Zhou, M., Wong, K.-F., Zhang, H.-J. (eds.) AIRS 2004. LNCS, vol. 3411. Springer, Heidelberg (2005)
17. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1), 96–118 (2007)
18. Qiu, Z., Zhao, X., Cai, C., Yang, H.: Towards the theoretical foundation of choreography. In: Proc. of WWW 2007, Banff, Canada. ACM Press, New York (2007)
19. van der Aalst, W., Dumas, M., Ouyang, C., Rozinat, A., Verbeek, H.: Choreography conformance checking: An approach based on BPEL and Petri Nets (extended version). Technical report, BPM Center Report BPM-05-25, BPMcenter.org (2005)
20. Yang, H., Cai, C., Peng, L., Zhao, X., Qiu, Z.: Reasoning about channel passing in choreography. In: TASE 2008. 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 135–142. IEEE Computer Society Press, Los Alamitos (2008)

Appendix: Proofs of Some Theorems

Consider conversations without "|" and local activities, thus a conversation is a sequence of interactions connected by ";". For the ease of discussion, we add incremental identifiers $I_1, ..., I_m$ for interactions in a conversation from beginning to end. The identifiers are also projected into peers' process.

Lemma 1. *For a conversation satisfies Condition 3 we proposed in Subsection 4.2, for any interaction I_h where peer r^i will use c (either transfer c or transfer through c), then $c \in C^i_{\mathsf{k}}$, or there is a sequence of interactions $I_{h_1}, ..., I_{h_s}$, where $h_1 < ... < h_s < h$, c is known by the sender of I_{h_1} initially, I_{h_j} $(1 \leq j < s-1,)$transfers c to the sender of $I_{h_{j+1}}$, and the receiver of I_{h_s} is r^i.*

Proof: It is directly implied by the function ϕ proposed in Section 2. □

Proof for **Theorem 3**: Apply induction on h, the subscript of interaction. Suppose interaction I_h is $c_1 : r^i \xrightarrow{c_2} r^j$, and $\theta_0(c_1) = y$, $\theta_0(c_2) = x$ where θ_0 is the injective mapping used in projection. If $h = 1$ then the behavior of the conversation must be $I_1; A$, $c_1 \in C^i_{\mathsf{k}}$. Thus peer $r^i = I_1 : c_1!c_2; \pi(A, i)$. peer $r^j = I_1 : c_1?x; \pi(A, j)$. Obviously, r^j will receive c_2 in I_1 and I_1 will terminate.

Suppose I_h, $h < k$ has terminated correctly, to prove I_k will terminate correctly. Suppose the conversation is $A; I_k; A'$ The configuration of the sender r^i

is $(\sigma^i, I_k : y!x; \pi(A', i))$. By Lemma 1, both c_1 and c_2 are known by r^i initially, or a sequence of finished interactions transfer them to r^i. We have $\sigma^i(y) = c_1$ and $\sigma^i(x) = c_2$.

At that time, the configuration of the receiver r^j must be $(\sigma^j, I_k : c_1!x; \pi(A', j))$. By the transition rule proposed in Section 3, I_k happens and r^j assigns channel variable x with c_2. □

Proof for **Theorem 4**: Because "|" and local activities would never affect the path for channel passing, Theorem 3 can be extended in the case that parallel and local activities are included. If all interactions will be carried out, then the system never deadlocks. Theorem 4 is proved. □

A Process Semantics for BPMN

Peter Y.H. Wong and Jeremy Gibbons

Computing Laboratory, University of Oxford, United Kingdom
{peter.wong,jeremy.gibbons}@comlab.ox.ac.uk

Abstract. Business Process Modelling Notation (BPMN), developed by the Business Process Management Initiative (BPMI), intends to bridge the gap between business process design and implementation. However, the specification of the notation does not include a formal semantics. This paper shows how a subset of the BPMN can be given a process semantics in Communicating Sequential Processes. Such a semantics allows developers to formally analyse and compare BPMN diagrams. A simple example of a business process is included to demonstrate the application of the semantics; some theoretical results about the semantics are briefly discussed.

1 Introduction

Modelling of business processes and workflows is an important area in software engineering. Business Process Modelling Notation (BPMN) [13] allows developers to take a process-oriented approach to modelling of systems. There are currently around forty implementations of the notation, but the notation specification developed by BPMI and adopted by OMG does not have a formal behavioural semantics, which we believe is crucial in behavioural specification and verification activities.

BPMN has been specified to map directly to the BPML standard, which has subsequently been superceded by WS-BPEL [2]. To the best of our knowledge the only previous attempt at defining a formal semantics for a subset of BPMN did so using Petri nets [4,5]. However, their semantics does not properly model multiple instances and does not allow comparisons of diagrams via refinements. A significant amount of work has been done towards the mapping between a particular class of BPMN diagrams and WS-BPEL [14,15], and the formal semantics of WS-BPEL [8,10,11,12]. However, as the use of graphical notations to assist the development process of complex software systems has become increasingly important, it is necessary to define a formal semantics for BPMN to ensure precise specification and to assist developers in moving towards correct implementation of business processes. A formal semantics also encourages automated tool support for the notation.

The main contribution of our work is to provide a formal process semantics for a subset of BPMN, in terms of the process algebra CSP [16]. By using the language and the behavioural semantics of CSP as the denotational model, we show how the existing refinement orderings defined upon CSP processes can be applied to the refinement of business process diagrams, and hence demonstrate

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 355–374, 2008.

how to specify behavourial properties using BPMN. Moreover, our processes may be readily analysed using a model checker such as FDR [7]. Our semantic construction starts from syntax expressed in Z [19], following Bolton and Davies's work on UML activity graphs [1].

This paper begins with an introduction to BPMN and the mathematical notations, Z [19] and CSP [16], that are used throughout the paper. Our contribution starts in Section 3, with a Z model of BPMN syntax, and continues in Section 4 with a behavioural semantics in CSP. In Section 5 we give a simple example to show how our semantics allows *consistency* between different levels of abstraction to be verified, and discuss briefly some theoretical results. We conclude this paper with a summary.

2 Notation

2.1 BPMN

States in our subset of BPMN [13] can either be pools, tasks, subprocesses, multiple instances or control gateways; they are linked by sequence, exception or message flows; sequence flows can be either incoming to or outgoing from a state and have associated guards; an exception flow from a state represents an occurrence of error within the state. Message flows represent directional communication between states. A sequence of sequence flows hence represents a specific control flow instance of the business process.

A table showing each type of state is presented in Figure 1. In the figure, a *start* state models the start of the business process in the current scope by initiating its outgoing transition. It has no incoming transition and only one outgoing transition. There are two types of end states *end* and *abort*. An *end*

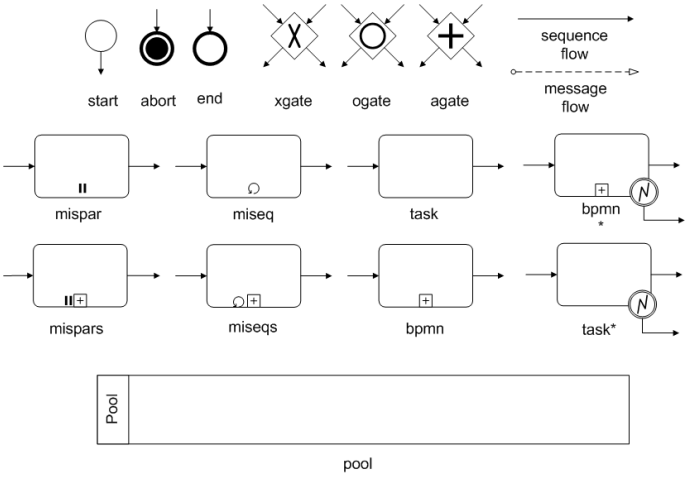


Fig. 1. States of BPMN diagram

state models the successful termination of an instance of the business process in the current scope by initialisation of its incoming transition. It has only one incoming transition with no outgoing transition. The *abort* state is a variant end state and models an unsuccessful termination, usually an error of an instance of the business process in the current scope.

Also in the figure, each of the *xgate*, *agate* and *ogate* state types has one or more incoming sequence flows and one or more outgoing sequence flows. An *xgate* state is an exclusive gateway, accepting one of its incoming flows and taking one of its outgoing flows; the semantics of this gateway type can be described as an exclusive choice and a simple merge. An *agate* state is a parallel gateway, which waits for all of its incoming flows before initialising all of its outgoing flows. An *ogate* state is an inclusive gateway, accepting one or more incoming sequence flows depending on their associated guards and initialising one or more of its outgoing flows also depending on their associated guards.

A *task* state describes an atomic activity and has exactly one incoming and one outgoing transition. A *bpmn* state describes a subprocess state; it is a business process by itself and so it models a flow of BPMN states. Figure 1 depicts a collapsed subprocess state where all internal details are hidden; this state has exactly one incoming and one outgoing transition. Also in Figure 1 there are graphical notations labelled *task** and *bpmn**, which depict a task state and a subprocess state with an exception flow. Each task and subprocess can also be defined as *multiple instances*. There are two types of multiple instances in BPMN: The *miseq* state type represents serial multiple instances, where the specified task is repeated in sequence; in the *mipar* state type the specified task is repeated in parallel. The types *miseqs* and *mipars* are their subprocess counterparts.

The graphical notation *pool* in Figure 1 depicts a participant within a business collaboration involving multiple business processes. Each pool forms a container for some business processes; only one process instance is allowed at any one time. While *sequence flows* are restricted to an individual pool, *message flows* represent communications between pools. For reasons of space we have omitted the syntactic and semantic definitions of message flows, details of which are in an extended version of this paper [18].

2.2 Z

The Z notation [19] has been widely used for state-based specification. It is based on typed set theory coupled with a structuring mechanism: the schema. A schema is essentially a pattern of declaration and constraint. Schemas may be named using the following syntax:

Name

declaration

constraint

or equivalently

$$Name \mathrel{\widehat=} [declaration \mid constraint]$$

If S is a schema then θS denotes the characteristic binding of S in which each component is associated with its current value. Schemas can be used as declarations. For example, the lambda expression $\lambda S \bullet t$ denotes a function from the schema type underlying S, a set of bindings, to the type of term expression t.

The mathematical language within Z provides a syntax for set expressions, predicates and definitions. Types can either be basic types, maximal sets within the specification, each defined by simply declaring its name, or be free types, introduced by identifying each of the distinct members, introducing each element by name. An alternative way to define an object within an specification is by abbreviation, exhibiting an existing object and stating that the two are the same.

$$Type ::= element_1 \mid ... \mid element_n \qquad [Type] \qquad symbol == term$$

By using an axiomatic definition we can introduce a new symbol x, an element of S, satisfying predicate p.

$$\begin{array}{|l} x : S \\ \hline p \end{array}$$

2.3 CSP

In CSP [16], a process is a pattern of behaviour; a behaviour consists of events, which are atomic and synchronous between the environment and the process. The environment in this case can be another process. Events can be compound, constructed using the dot operator '.'; often these compound events behave as channels communicating data objects synchronously between the process and the environment. Below is the syntax of the language of CSP.

$$\begin{aligned} P, Q ::= {} & P \mid\mid\mid Q \mid P \,[\![A]\!]\, Q \mid P \,[\![A \mid B]\!]\, Q \mid P \setminus A \mid P \triangle Q \mid \\ & P \Box Q \mid P \sqcap Q \mid P \fatsemi Q \mid e \rightarrow P \mid Skip \mid Stop \\ e \quad ::= {} & x \mid x.e \end{aligned}$$

Process $P \mid\mid\mid Q$ denotes the interleaved parallel composition of processes P and Q. Process $P \,[\![A]\!]\, Q$ denotes the partial interleaving of processes P and Q sharing events in set A. Process $P \,[\![A \mid B]\!]\, Q$ denotes parallel composition, in which P and Q can evolve independently but must synchronise on every event in the set $A \cap B$; the set A is the alphabet of P and the set B is the alphabet of Q, and no event in A and B can occur without the cooperation of P and Q respectively. We write $|||\, i : I \bullet P(i)$, $||[A]\, i : I \bullet P(i)$ and $||\, i : I \bullet A(i) \circ P(i)$ to denote an indexed interleaving, partial interleaving and parallel combination of processes $P(i)$ for i ranging over I.

Process $P \setminus A$ is obtained by hiding all occurrences of events in set A from the environment of P. Process $P \triangle Q$ denotes a process initially behaving as P,

but which may be interrupted by Q. Process $P \,\Box\, Q$ denotes the external choice between processes P and Q; the process is ready to behave as either P or Q. An external choice over a set of indexed processes is written $\Box\, i : I \bullet P(i)$. Process $P \sqcap Q$ denotes the internal choice between processes P or Q, ready to behave as at least one of P and Q but not necessarily offer either of them. Similarly an internal choice over a set of indexed processes is written $\sqcap\, i : I \bullet P(i)$.

Process $P \fatsemi Q$ denotes a process ready to behave as P; after P has successfully terminated, the process is ready to behave as Q. Process $e \rightarrow P$ denotes a process capable of performing event e, after which it will behave like process P. The process *Stop* is a deadlocked process and the process *Skip* is a successful termination.

CSP has three denotational semantics: traces ($\mathcal{T}$), stable failures ($\mathcal{F}$) and failures-divergences ($\mathcal{N}$) models, in order of increasing precision. In this paper our process definitions are divergence-free, so we will concentrate on the stable failures model. The traces model is insufficient for our purposes, because it does not record the availability of events and hence only models what a process can do and not what it must do [16]. For example, the processes $a \rightarrow \mathit{Skip}$ and $(a \rightarrow \mathit{Skip}) \sqcap \mathit{Stop}$ have the same traces (the traces model is prefix-closed), even though the latter one is allowed to do nothing at all no matter what we offer it. In order to distinguish these processes, it is necessary to record not only what a process can do, but also what it can refuse to do. This information is preserved in *refusal sets*, sets of events from which a process in a stable state can refuse to communicate no matter how long it is offered. The set $\mathit{refusals}(P)$ is P's initial refusals. A failure therefore is a pair (s, X) where $s \in \mathit{traces}(P)$ is a trace of P leading to a stable state and $X \in \mathit{refusals}(P/s)$ where P/s represents process P after the trace s. We write $\mathit{traces}(P)$ and $\mathit{failures}(P)$ as the set of all P's traces and failures respectively.

We write Σ to denote the set of all event names, and *CSP* to denote the syntactic domain of process terms. We define the semantic function $\mathcal{F}$ to return the set of all traces and the set of all failures of a given process, whereas the semantic function $\mathcal{T}$ returns solely the set of traces of the given process.

$$\mathcal{F} : \mathit{CSP} \rightarrow (\mathbb{P} \operatorname{seq} \Sigma \times \mathbb{P}(\operatorname{seq} \Sigma \times \mathbb{P}\, \Sigma))$$
$$\mathcal{T} : \mathit{CSP} \rightarrow \mathbb{P} \operatorname{seq} \Sigma$$

These models admit refinement orderings based upon reverse containment; for example, for the stable failures model we have

$$\_ \sqsubseteq_{\mathcal{F}} \_ : \mathit{CSP} \leftrightarrow \mathit{CSP}$$
$$\forall\, P, Q : \mathit{CSP} \bullet$$
$$P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathit{traces}(P) \supseteq \mathit{traces}(Q) \land \mathit{failures}(P) \supseteq \mathit{failures}(Q)$$

While traces only carry information about *safety* conditions, refinement under the stable failures model allows one to make assertions about a system's *safety* and *availability* properties. These assertions can be automatically proved using a model checker such as FDR [7], exhaustively exploring the state space of

a system, either returning one or more counterexamples to a stated property, guaranteeing that no counterexample exists, or until running out of resources.

3 Syntactic Description of BPMN

In this section we describe the abstract syntax of BPMN using Z schemas and set theory, and use an example in Section 3.2 to show how the syntax can be applied on a given BPMN diagram. For reasons of space, we have omitted certain schema and function definitions and have only concentrated on the definition of a smaller subset of the BPMN states than shown in Section 2; readers may refer to our longer paper [18] for their full definitions.

3.1 Abstract Syntax

We first introduce some maximal sets of values to represent constructs such as *lines*, *task* and *subprocess name*, defined as Z's basic types:

$$[CName, PName, Task, Line, Guard]$$

We then derive subtypes $BName$ and $PLName$ axiomatically:

$$BName, PLName : \mathbb{P}\, PName$$
$$\langle BName, PLName \rangle \text{ partition } PName$$

The sequence of sets $\langle S_1 \,..\, S_n \rangle$ *partitions* some set T iff

$$\bigcup S_1 \,..\, S_n = T \land (\forall\, i, j : 1 \,..\, n \bullet S_i \cap S_j = \emptyset)$$

Each type of state shown in Figure 1 is defined using the free type $Type$ where each of its constructors describes a particular type of states. For example, the type of an atomic *task* state is defined by $task\, t$ where t is a unique name that identifies that task state. Below is the partial definition.

$$\begin{aligned} Type ::= {} & start \mid end\langle\langle \mathbb{N} \rangle\rangle \mid abort\langle\langle \mathbb{N} \rangle\rangle \mid task\langle\langle Task \rangle\rangle \mid \\ & xgate \mid bpmn\langle\langle BName \rangle\rangle \mid miseq\langle\langle Task \times \mathbb{N} \rangle\rangle \end{aligned}$$

According to the specification [13], each BPMN state type has other associated attributes describing its properties; our syntactic definition has included only some of these attributes. For example, the number of loops of a sequence multiple instance state type is recorded by the natural number in the constructor function *miseq*. We define some abbreviations as follows to assist our specification.

$$Tasks == \text{ran}\ task \cup \text{ran}\ miseq \cup \text{ran}\ mipar$$
$$Subs == \text{ran}\ bpmn \cup \text{ran}\ mipars \cup \text{ran}\ miseqs$$

In this paper we call both sequence flows and exception flows 'transitions'; states are linked by transition lines representing flows of control, which may have associated guards. We give the type of a sequence flow or an exception flow by the following schema definition.

$$Trans \mathrel{\widehat{=}} [guard : Guard;\ line : Line]$$

Here we show a partial definition of the schema *State* for each BPMN *state*, omitting the inclusion of schema components for message flows.

$$State \mathrel{\widehat=} [type : Type;\ in, out, error : \mathbb{P}\, Trans;\ exit : \mathbb{P}(\mathbb{N} \times Trans);\ loop : \mathbb{N}]$$

Each state records the type of its content, the sets of incoming, outgoing and error transitions, and in the case of a subprocess state, a set of number-transition pairs to align the outgoing transitions of the subprocess within the outgoing transitions within the subprocess. Each state incorporates the variable *loop* to limit the number of state instances the process instance can invoke. The state also records different types of message flows, but we have omitted their definition in this paper.

We denote a subset of *well-formed* states in BPMN by the schema type *WFS*, and we define the type $WCF : \mathbb{P}(\mathbb{P}\, State)$ to be the set of *well-configured* sets of well-formed states *WCF*. Well-formedness is defined to conform to the constraints within the official documentation [13]; for example, a *start state* must have no incoming transition and only one outgoing transition. A definition of this subset may be found in the extended version of this paper [18].

Each BPMN diagram, encapsulated by a *pool* representing an individual participant in a collaboration, is built up from a well-configured finite set of well-formed states. We do not allow local states to have type *pool*, since this represents a boundary of a business domain. The function type *Local* represents the environment of the local specification and each function of its type maps each name of a BPMN diagram to its associated diagram. Consequently a collaboration is built up from a finite set of names, and each of the names is associated with a BPMN diagram. For reasons of space both the syntactic and semantic definition of collaboration have been omitted, again, see the extended version of this paper [18].

$$BPD ::= states\langle\!\langle WCF \rangle\!\rangle$$
$$Local == PName \nrightarrow BPD$$

3.2 An Example

We present an example of a business process of an airline reservation system shown in Figure 2; this example has been taken from the WSCI specification [17]. It could be assumed to have been constructed during the development of the reservation system. We have abstracted message flows, as there is only one business participant in the example. We use this example to illustrate how a BPMN diagram can be translated into a well-configured set of states describing the diagram's syntax.

We observe that the airline reservation business process is initiated by verifying seat availability, after which seats may be reserved. If the reservation period elapses, the business process will cancel the reservation automatically and notify the user. The user might decide to cancel her reservation, or proceed with the booking. Upon a successful booking, tickets will be issued.

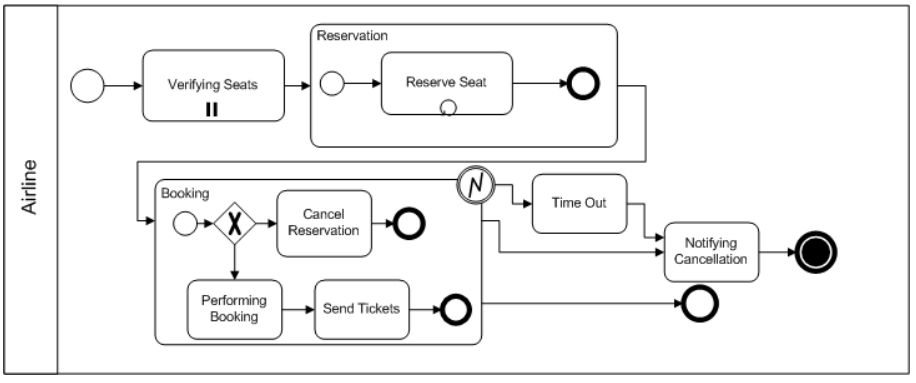


Fig. 2. A BPMN diagram describing the workflow of an airline reservation application

Given the business process name *airline*, the following shows a set of well-formed states translated from the diagram describing the reservation part of business process. We have omitted details of the bindings of *Trans* and *Messageflow*. We write $a_1 \mathinner{..} a_n \rightsquigarrow \emptyset$ inside some schema binding s to specify the components $s.a_1 \mathinner{..} s.a_n$ to be empty. The syntactic details of the subprocesses *Reserve* and *Booking* are also omitted.

$$
\begin{array}{|l}
airline : PName;\ book, reserve : BName;\ verify, timeout, notify : Task \\
\hline
\exists\, local : Local;\ t1, t2, t3, t4, t5, t6, t7, t8 : Trans;\ i, j, k, l, m, n : \mathbb{N} \bullet \\
\quad states^{\sim}(local\ airline) = \\
\qquad \{ \langle type \rightsquigarrow start, out \rightsquigarrow \{ t1 \}, in, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow mipar\ verify\ n, in \rightsquigarrow \{ t1 \}, out \rightsquigarrow \{ t2 \}, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow bpmn\ reserve, in \rightsquigarrow \{ t2 \}, out \rightsquigarrow \{ t3 \}, error \rightsquigarrow \emptyset, exit \rightsquigarrow \{ (m, t3) \} \rangle, \\
\qquad\ \langle type \rightsquigarrow bpmn\ book, in \rightsquigarrow \{ t3 \}, out \rightsquigarrow \{ t4, t5 \}, error \rightsquigarrow \{ t6 \}, \\
\qquad\qquad exit \rightsquigarrow \{ (k, t4), (l, t5) \} \rangle, \\
\qquad\ \langle type \rightsquigarrow task\ timeout, in \rightsquigarrow \{ t6 \}, out \rightsquigarrow \{ t7 \}, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow task\ notify, in \rightsquigarrow \{ t5, t7 \}, out \rightsquigarrow \{ t8 \}, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow end\ i, in \rightsquigarrow \{ t4 \}, out, error, exit \rightsquigarrow \emptyset \rangle, \\
\qquad\ \langle type \rightsquigarrow abort\ j, in \rightsquigarrow \{ t8 \}, out, error, exit \rightsquigarrow \emptyset \rangle\ \}
\end{array}
$$

4 Behavioural Semantics of BPMN

In Section 3 we gave an overview of the abstracted syntax for BPMN in Z. In this section, we define a semantic function which takes the syntactic description of a BPMN diagram and returns the CSP process that models the behaviour of that diagram. That is, the function returns the parallel composition of processes corresponding to the states of the diagram, each synchronising on its own alphabet, which represents its transition events, to ensure the correct order of control flow. For reasons of space, we only consider the semantics of a BPMN

diagram with a single participant (i.e. one pool), and each function associated to the semantics will be defined over a smaller subset of the BPMN states, namely the states of type *start*, *end*, *task*, *miseq*, *miseqs* (subprocess), *bpmn* (subprocess), *agate*, *xgate* and *ogate*, which have been described in Section 2; the semantics of other states in the figure may be defined similarly. The complete semantic definition of business collaboration and of other states may be found in our longer paper [18]. The rest of the section is structured as follows: in Section 4.1 we define functions to associate each transition, state and diagram with their set of events; Section 4.2 presents the overall semantic functions for mapping each BPMN diagram to its process describing its behaviour; in Section 4.3 we present the CSP processes corresponding to the behaviour of each gateway; and in Section 4.4 we define processes corresponding to the behaviour of each state type and transition, and the general functions for mapping each BPMN state to the CSP process describing its behaviour.

4.1 Alphabets

First we define the basic types *Process* and *Event* which correspond to CSP processes and events.

$$[Process, Event]$$

We define the partial injective function ϵ_{trans} which maps each transition to a pair of a CSP event and a guard. We insist that each transition maps to a unique CSP event. The functions ϵ_{task} and ϵ_{pname} map each task and process name to a unique event respectively.

$$\begin{array}{l} \epsilon_{line} : Line \rightarrowtail Event \\ \epsilon_{task} : Task \rightarrowtail Event \\ \epsilon_{pname} : PName \rightarrowtail Event \\ \epsilon_{trans} : Trans \rightarrowtail (Event \times Guard) \\ \hline \epsilon_{trans} = \lambda\, Trans \bullet (\epsilon_{line}\, line, guard) \end{array}$$

In order to define the alphabet for each state, corresponding to the events on which each state must synchronise, we must consider the events associated with each transition, type and messageflow. We define the function α_{trans} which maps each set of transitions to the set of associated events. (Given a tuple of n elements t, we use the projection notation $t.m$ to the denote the mth element of the tuple.)

$$\begin{array}{l} \alpha_{trans} : \mathbb{P}\, Trans \rightarrowtail \mathbb{P}\, Event \\ \hline \alpha_{trans} = \lambda\, ts : Trans \bullet \{\, t : \epsilon_{trans}(\!|\, ts \,|\!) \bullet t.1 \,\} \end{array}$$

The alphabet of a given state is the set of events associated with a state with which it must synchronise. A state's alphabet is the union of the events mapped from all its incoming and outgoing transitions, type and exception flows. We define α_{state} to be a function mapping each state into its alphabet.

$$\alpha_{state} : Local \nrightarrow State \nrightarrow \mathbb{P}\, Event$$

$$\begin{aligned}
&\alpha_{state} = (\lambda\, l : Local \bullet (\lambda\, State \bullet \\
&\textbf{if } (type \notin (Tasks \cup Subs)) \textbf{ then } \alpha_{trans}(out \cup in) \\
&\textbf{else } (\textbf{if } (type \in \operatorname{ran} miseq) \textbf{ then } \alpha_{trans}\,(\mu\, t : miseqt\, s \bullet \{\, t.1, t.2 \,\}) \textbf{ else } \emptyset)) \\
&\qquad \cup (\textbf{if } (type \in Subs) \textbf{ then } \bigcup((\alpha_{state}\, l) \lhd states^{\sim}(l(bpmn^{\sim} type))) \\
&\qquad\quad \textbf{else } (\textbf{if } (type \notin Tasks) \textbf{ then } \emptyset \textbf{ else } \{\, \epsilon_{task}(task^{\sim} type) \,\})) \\
&\qquad \cup \alpha_{trans}(out \cup in \cup error))))
\end{aligned}$$

The function *miseqt* maps each state of type *miseq* to a transition pair used to connect the state's task or subprocess state.

$$miseqt : State \rightarrowtail (Trans \times Trans)$$

$$miseqt = (\lambda\, State \bullet (\mu (s, t) : (Trans \times Trans) \mid s \neq t))$$

We also define the function $\alpha_{process}$ to map each diagram to the set of all possible events performed by the process describing an individual *local* diagram's behaviour.

$$\alpha_{process} : PName \nrightarrow Local \nrightarrow \mathbb{P}\, Event$$

$$\begin{aligned}
&\forall\, p : PName;\ local : Local \bullet \\
&\qquad \alpha_{process} = \bigcup\{\, s : states^{\sim}(local\ p) \bullet \alpha_{state}\ s\ local \,\}
\end{aligned}$$

4.2 Processes Corresponding to Diagrams

Our semantics abstracts the internal flow of individual task states and only models the sequence of task initialisations and terminations within a business process. Our semantic function *bsem* takes a syntactic description of a BPMN diagram encapsulated by a state of type *pool* or a BPMN subprocess and returns a parallel composition of processes, each corresponding to one of the diagram's or process's states. The parallel composition, defined by the function *bsm*, is conjoined via partial interleaving with process X to ensure that the business process either terminates successfully or deadlocks because of an exception flow. We define compound events $fin.i$ and $abt.i$ where i ranges over $\mathbb{N}$ to denote the successful completion and the abortion of a business process.

$$\begin{aligned}
&bsem : PName \nrightarrow Local \nrightarrow Process \\
&hide : PName \nrightarrow Local \nrightarrow \mathbb{P}\, Event
\end{aligned}$$

$$\begin{aligned}
&\forall\, p : PName;\ l : Local \bullet \\
&bsem\ p\ l = \\
&\qquad \textbf{let } A = \{a : \epsilon_{abort}\ p\ l;\ e : \epsilon_{end}\ p\ l \bullet fin.e, abt.a\,\} \cup \alpha_{proc}\ p\ l \\
&\qquad\quad X = \Box\, i : \alpha_{proc}\ p\ l \bullet i \rightarrow X \ \Box\ (\Box\, e : \epsilon_{abort}\ p\ l \bullet abt.e \rightarrow Stop) \\
&\qquad\qquad \Box\ (\Box\, e : \epsilon_{end}\ p\ l \bullet fin.e \rightarrow Skip) \\
&\qquad \textbf{in } \ (bsm\ p\ l \,[\![\, A \,]\!]\, X) \setminus hide\ p\ l \\
&\wedge\ hide\ p\ l = \bigcup\{\, s : states^{\sim}(l\ p) \bullet \alpha_{trans}(s.in \cup s.out \cup s.error) \,\}
\end{aligned}$$

$$
\begin{array}{l}
bsm : PName \nrightarrow Local \nrightarrow Process \\
\hline
\forall\, p : PName;\ l : Local \bullet \\
bsm\ p\ l = \\
\quad (\ \Big\|\ s : \{\, s : (states^{\sim}(l\ p)) \mid s.type \neq start \,\} \bullet \\
\qquad (\alpha_{state}\ s\ l \cup \{\, i : \epsilon_{end}\ p\ l \bullet fin.i \,\} \\
\qquad \cup (\textbf{if}\ (s.type \notin \operatorname{ran} abort)\ \textbf{then}\ \emptyset\ \textbf{else}\ \{\, abt.(abort^{\sim} s.type) \,\}) \circ \\
\qquad \textbf{if}\ (s.type \in \operatorname{ran} end) \\
\qquad \textbf{then}\ ((\rho_{state}\ s \fatsemi fin.(end^{\sim} s.type) \rightarrow Skip) \\
\qquad\qquad \Box\ (\Box\ e : \epsilon_{end}\ p\ l \setminus \{\, end^{\sim} s.type \,\} \bullet fin.e \rightarrow Skip)) \\
\qquad \textbf{else if}\ (s.type \in \operatorname{ran} abort) \\
\qquad\qquad \textbf{then}\ ((\rho_{state}\ s \fatsemi abt.(abort^{\sim} s.type) \rightarrow Stop) \Box\ \rho_{end}\ p\ l) \\
\qquad\qquad \textbf{else let}\ X = ((\rho_{state}\ s \Box\ \rho_{end}\ p\ l)\ \textbf{in} \\
\qquad\qquad\qquad (\textbf{if}\ s.loop = 0\ \textbf{then}\ X \\
\qquad\qquad\qquad \textbf{else}\ (\rho_{loop}\ p\ s\ l\ [\![\alpha_{trans}\ s.in \cup \{\, i : \epsilon_{end}\ p\ l \bullet fin.i \,\}]\!]\ X)))) \\
\quad [\![\alpha_{start}\ p\ l \cup \{\, i : \epsilon_{end}\ p\ l \bullet fin.i \,\}]\!] \\
\quad \Box\ s : \{\, s : states^{\sim}(l\ p) \mid s.type = start \,\} \bullet (\rho_{state}\ s \fatsemi \rho_{end}\ p\ l))
\end{array}
$$

We observe that the processes corresponding to a start, an end or an abort state are the only non-recursive processes; a start, an end or an abort activity can occur only once, while it is possible for all other states to occur many times within a single process instance. The function ϵ_{end} returns the set of numbers defined by each of the *end* states within the diagram's syntax, while ϵ_{abort} returns the set of numbers defined by each of the *abort* states. We apply external choice over the processes corresponding to states with a terminating process synchronising on all *end* states. This ensures that all processes terminate at the end of the business process execution. The function α_{start} returns the set of events corresponding to all outgoing transitions of all *start* states within the diagram's syntax.

$$
\begin{array}{l}
\alpha_{start} : PName \nrightarrow Local \nrightarrow \mathbb{P}\, Event \\
\epsilon_{end} : PName \nrightarrow Local \nrightarrow \mathbb{P}\, \mathbb{N} \\
\hline
\forall\, p : PName;\ local : Local \bullet \\
\qquad \alpha_{start}\ p\ local = \bigcup \{\, s : states^{\sim}(local\ p) \mid s.type = start \bullet \alpha_{trans}(s.out) \,\} \\
\qquad \wedge\ \epsilon_{end}\ p\ local = \{\, s : states^{\sim}(local\ p) \mid s.type \in \operatorname{ran} end \bullet end^{\sim} s.type \,\}
\end{array}
$$

$$
\begin{array}{l}
\rho_{end} : PName \nrightarrow Local \nrightarrow Process \\
\epsilon_{abort} : PName \nrightarrow Local \nrightarrow \mathbb{P}\, \mathbb{N} \\
\hline
\forall\, p : PName;\ local : Local \bullet \\
\qquad \rho_{end}\ p\ local = (\Box\ e : \epsilon_{end}\ p\ local \bullet fin.e \rightarrow Skip)) \\
\qquad \wedge\ \epsilon_{abort}\ p\ local = \\
\qquad\qquad \{\, s : states^{\sim}(local\ p) \mid s.type \in \operatorname{ran} abort \bullet abort^{\sim} s.type \,\} \\
\qquad\qquad \cup \bigcup \{\, s : states^{\sim}(local\ p) \mid s.type \in \operatorname{ran} bpmn \bullet \\
\qquad\qquad\qquad\qquad \epsilon_{abort}\ (bpmn^{\sim} s.type)\ local \,\}
\end{array}
$$

The function ρ_{loop} maps each state of type *task* and *bpmn* to a process which limits the number of iterations of the state.

$$\rho_{loop} : PName \nrightarrow State \nrightarrow Local \nrightarrow Process$$

$$\begin{aligned}
&\forall\, p : PName;\ s : State;\ local : Local \bullet \\
&\rho_{loop}\ p\ s\ local = \\
&\quad \textbf{let}\ Y = \Box\, i : \alpha_{trans}\ s.in \bullet i \rightarrow Skip \\
&\qquad M = \rho_{extmsg}\ s.in\ NoEnds \\
&\qquad X(n) = n > 0\ \&\ (Y \fatsemi X(n-1) \Box (M \fatsemi Y \fatsemi X(n-1)) \Box \rho_{end}\ p\ local) \\
&\qquad\qquad \Box\ n \leq 0\ \&\ \rho_{end}\ p\ local \\
&\quad \textbf{in}\ X(loopMax)
\end{aligned}$$

We define the function ρ_{miseq} to map each state of type *miseq* or *miseqs*. The following describes the function ρ_{miseq}.

$$\rho_{miseq} : State \nrightarrow Local \nrightarrow Process$$

$$\begin{aligned}
&\forall\, s : State;\ local : Local \bullet \exists\, t1, t2 : Trans;\ e1, e2 : Event;\ n : \mathbb{N} \bullet \\
&\quad (t1, t2) = miseqtst\ s \wedge (e1, e2) = ((\epsilon_{trans}\ t1).1, (\epsilon_{trans}\ t2).1) \\
&\quad \wedge\ (\textbf{if}\ s.type \in \operatorname{ran} miseq\ \textbf{then}\ n = (miseq^{\sim} s.type).2\ \textbf{else}\ n = (miseqs^{\sim} s.type).2) \\
&\quad \wedge\ \rho_{miseq}\ s\ local = \\
&\qquad\qquad \textbf{let}\ \ SY = \alpha_{trans}(s.out \cup s.error) \cup \{\, e1, e2 \,\} \\
&\qquad\qquad \textbf{in}\ \ ((Cq(n, s, e1, e2) \,[\![\, SY \,]\!]\, Seq(n, s, local)) \triangle AJ(s.error)) \setminus \{\, e1, e2 \,\}
\end{aligned}$$

The function ρ_{miseq} is constructed by partially interleaving a control process *Cq* with process *Seq*, which models the multiple instances of task or subprocess, specified by the contructor function, executing sequentially.

$$\begin{aligned}
&Seq(i, s, l) = \\
&\quad \textbf{let}\ \ tpe = \textbf{if}\ s.type \in \operatorname{ran} miseq\ \textbf{then}\ task\ (miseq^{\sim} s.type)\ \textbf{else}\ bpmn\ (miseqs^{\sim} s.type) \\
&\qquad\ st = \langle\!|\, in \rightsquigarrow \{\, t1 \,\}, type \rightsquigarrow tpe, out \rightsquigarrow \{\, t2 \,\}, error \rightsquigarrow s.error, loop \rightsquigarrow 1 \,|\!\rangle \\
&\quad \textbf{in}\ i > 0\ \&\ ((\rho_{state}\ st\ l) \fatsemi Seq(i-1, s, l)) \Box XS(s.out)
\end{aligned}$$

The process *Cq* is triggered initially by one of the incoming transitions of the multiple instance state. Instances are triggered sequentially.

$$\begin{aligned}
&Cq(n, s, e, f) = \\
&\quad ((XJ(s.in) \Box f \rightarrow Skip) \fatsemi \\
&\quad ((n > 1)\ \&\ (e \rightarrow Cq(n-1, s, e, f)) \Box n = 1\ \&\ (e \rightarrow f \rightarrow XS(s.out)) \Box XS(s.out)))
\end{aligned}$$

4.3 Processes Corresponding to Gateways

We now define some CSP processes that correspond to the behaviour of each of the gateway states.

Exclusive Choice Gateway. Processes $XS(tn)$ and $XJ(tn)$ model the behaviour of outgoing and incoming transitions of the state type *xgate*. Note that although each outgoing transition of the state type *xgate* is guarded, the choice of its incoming transitions is determined by the behaviour of the preceding states.

$$\begin{aligned}
XS(tn) &= \Box\, e : \epsilon_{trans}(\!|\ tn\ |\!) \bullet\ (\text{if } e.2 \text{ then } e.1 \rightarrow Skip \text{ else } Skip) \\
XJ(tn) &= \Box\, e : \alpha_{trans}\ tn \bullet e \rightarrow Skip
\end{aligned}$$

We also define the process $AJ(tn)$ to model the behaviour of incoming transitions of the state type *abort*.

$$AJ(tn) = \Box\, e : \alpha_{trans}\ tn \bullet e \rightarrow Stop$$

Parallel Gateway. Process $ASJ(tn)$ models the behaviour of outgoing and incoming transitions of the state type *agate*. Note that all outgoing transitions are enabled and all incoming transitions are required in this state type.

$$ASJ(tn) = |||\, e : \alpha_{trans}\ tn \bullet e \rightarrow Skip$$

Inclusive Choice Gateway. Process $OSJ(tn)$ models the behaviour of outgoing and incoming transitions of the state type *ogate*. Note that all outgoing transitions are guarded in the state type *ogate*, one or more transitions are enabled and the choice of transitions is based on the value of their guards. All its incoming transitions are also guarded; the choice of transitions is based on the value of their guards.

$$OSJ(tn) = |||\, e : \epsilon_{trans}(\!|\ tn\ |\!) \bullet\ (\text{if } e.2 \text{ then } e.1 \rightarrow Skip \text{ else } Skip)$$

4.4 Processes Corresponding to Transitions, Types and States

Functions ρ_{out} and ρ_{in} take a state and return the process describing the behaviour of all outgoing and incoming transitions, respectively.

$$\begin{array}{l}
\rho_{out} : State \nrightarrow Process \\
\rho_{in} : State \nrightarrow Process \\
\hline
\rho_{out} = (\lambda\, State \bullet \ \textbf{if } (type = asplit) \textbf{ then } ASJ(out) \\
\qquad\qquad \textbf{else if } (type = osplit) \textbf{ then } OSJ(out) \textbf{ else } XS(out)) \\
\rho_{in} = (\lambda\, State \bullet \ \textbf{if } (type \in \operatorname{ran} abort) \textbf{ then } AJ(in) \\
\qquad\qquad \textbf{else if } (type = ajoin) \textbf{ then } ASJ(in) \\
\qquad\qquad\quad \textbf{else if } (type = ojoin) \textbf{ then } OSJ(in) \textbf{ else } XJ(in))
\end{array}$$

The function ρ_{type} maps the type of a given state to its corresponding process. Since our semantics abstracts the internal flow of task states, we only model the initialisation, the termination, message flows and any exception flow of each task.

$$\begin{array}{l}
\rho_{exit} : State \nrightarrow Process \\
\rho_{type} : State \nrightarrow Local \nrightarrow Process \\
\hline
\rho_{exit} = (\lambda\, State \bullet \\
\quad \textbf{let } Y = \{\, (e, f) : exit \bullet (fin.e, (\epsilon_{trans}\, f).1)\,\} \\
\quad \textbf{in } (\Box (i, j) : Y \bullet i \rightarrow j \rightarrow Skip) \,\Box\, XJ(error)) \\
\rho_{type} = (\lambda\, State \bullet (\lambda\, l : Local \bullet \\
\ \textbf{if } (type \in \operatorname{ran} task) \\
\ \textbf{then if } (error = \emptyset) \textbf{ then } \epsilon_{task}(task^{\sim} type) \textbf{ else } \epsilon_{task}(task^{\sim} type) \triangle XJ(error) \\
\ \textbf{else if } (type \notin \operatorname{ran} task \cup \operatorname{ran} bpmn) \textbf{ then } Skip \\
\quad \textbf{else } (\textbf{if } (error = \emptyset) \textbf{ then } \epsilon_{pname}(bpmn^{\sim} type) \rightarrow bsem\,(bpmn^{\sim} type)\, l \\
\qquad \textbf{else } \epsilon_{pname}(bpmn^{\sim} type) \rightarrow (bsem\,(bpmn^{\sim} type)\, l \triangle XJ(error)))))
\end{array}$$

We define the function ρ_{state} which returns the process corresponding to the behaviour of a given state; this function essentially maps each state to the sequential composition of the processes corresponding to the state's incoming transitions, type and outgoing transitions.

$$
\begin{array}{l}
\rho_{state} : State \nrightarrow Local \nrightarrow Process \\
\hline
\rho_{state} = (\lambda\, s : State \bullet (\lambda\, l : Local \bullet \\
\textbf{if } (type \in \operatorname{ran} task) \textbf{ then } (\rho_{in}\ s \fatsemi \rho_{type}\ s\ l \fatsemi \rho_{out}\ s) \\
\textbf{else if } (type \in \operatorname{ran} bpmn) \\
\quad \textbf{then } (\rho_{in}\ s \fatsemi ((\rho_{type}\ s\ l \,[\![\, \{\, e : exit \bullet fin.(e.1) \,\} \cup \alpha_{trans}\ error]\!] \\
\qquad \rho_{exit}\ s\ l) \,[\![\, \{\, o : out \bullet (\epsilon_{trans}\ e).1 \,\} \,]\!]\, \rho_{out}\ s)))) \\
\quad \textbf{else if } (type \in \operatorname{ran} miseq \cup \operatorname{ran} miseqs) \textbf{ then } \rho_{miseq}\ s\ l \\
\qquad \textbf{else if } (type = start) \textbf{ then } \rho_{out}\ s \\
\qquad\quad \textbf{else if } (type \in \operatorname{ran} end \cup \operatorname{ran} abort) \textbf{ then } \rho_{in}\ s \\
\qquad\qquad \textbf{else } \rho_{in}\ s \fatsemi \rho_{out}\ s))
\end{array}
$$

We have implemented the semantics described in this paper as a prototype tool using the functional programming language Haskell. Readers may find a copy of the implementation from our web site[1]. The tool inputs a XML serialised representation of BPMN diagram from the JViews BPMN Modeler [9], and translates it into an ASCII file containing CSP processes representing its behaviours expressed in machine-readable CSP [16].

5 Revisiting the Example

5.1 Semantics of the Airline Reservation Application

We use the example of an airline reservation system in Section 3.2 to demonstrate how our semantic function may be applied to the syntactic definition described in Section 3, and hence provide a semantics to support formal analyses. We define set J to index the processes corresponding to the states in the diagram.

$$J = \{\, start, verify, reserve, booking, notify, timeout, end, abort \,\}$$

By applying our semantic function to the diagram's syntactic description, we obtain the process corresponding to it.

$$
\begin{array}{rl}
Airline = \textbf{let } X = \Box\, i : (\alpha Y \setminus \{\, fin.1, abt.1 \,\}) \bullet & \\
\qquad\qquad\qquad\qquad (i \rightarrow X \ \Box\ abt.1 \rightarrow Stop \ \Box\ fin.1 \rightarrow Skip) & \\
\qquad\qquad Y = (\ \|\ j : J \bullet \alpha P(j) \circ P(j)) & \\
\qquad \textbf{in } (Y \,[\![\, \alpha Y \,]\!]\, X) \setminus \{\!|init|\!\} &
\end{array}
$$

where for each j in J, the process $P(j)$ is as defined below and $\alpha P(j)$ is the set of possible events performed by $P(j)$. We use n, ranging over $\mathbb{N}$, to denote the

[1] http://www.comlab.ox.ac.uk/peter.wong/observation/

number of instances of the task *verify*, as specified by the second argument of constructor function *miseq*.

$$
\begin{array}{l}
P(\mathit{verify}) = \\
\quad \textbf{let} \\
\qquad \mathit{Ts} = \{\, i : \{\, 1 \,..\, n \,\} \bullet (\mathit{in}.i, \mathit{out}.i) \,\} \\
\qquad \mathit{IC}(T) = \Box(i,j) : T \bullet i \rightarrow (j \rightarrow \mathit{Skip} \;|||\; \mathit{Cn}(T \setminus \{\, i,j \,\})) \\
\qquad \mathit{Cn}(T) = \#T = 1 \;\&\; (\Box(i,j) : T \bullet i \rightarrow j \rightarrow \mathit{init}.\mathit{reserve} \rightarrow \mathit{Skip}) \\
\qquad\qquad \Box\; \#T > 1 \;\&\; \mathit{IC}(T) \;\Box\; \mathit{init}.\mathit{reserve} \rightarrow \mathit{Skip} \\
\qquad \mathit{MTask} = \; \|[\{\, \mathit{init}.\mathit{reserve} \,\}](i,j) : \mathit{Ts} \bullet \\
\qquad\qquad ((i \rightarrow \mathit{starts}.\mathit{verify} \rightarrow j \rightarrow \mathit{Skip} \fatsemi \mathit{init}.\mathit{reserve} \rightarrow \mathit{Skip}) \;\Box\; \mathit{init}.\mathit{reserve} \rightarrow \mathit{Skip}) \\
\quad \textbf{in}\; ((\mathit{init}.\mathit{verify} \rightarrow \mathit{Skip} \fatsemi \\
\qquad\quad (\mathit{MTask} \;[\![\bigcup\{\, (i,j) : \mathit{Ts}\{\, i,j \,\} \,\} \cup \{\, \mathit{init}.\mathit{reserve} \,\}]\!] \\
\qquad\quad (\mathit{init}.\mathit{reserve} \rightarrow \mathit{Skip} \;\Box\; \mathit{Cn}(\mathit{Ts})))) \fatsemi P(\mathit{verify})) \;\Box\; \mathit{fin}.1 \rightarrow \mathit{Skip}
\end{array}
$$

$$
\begin{array}{rl}
P(\mathit{start}) = & (\mathit{init}.\mathit{verify} \rightarrow \mathit{Skip}) \fatsemi (\mathit{fin}.1 \rightarrow \mathit{Skip}) \\
P(\mathit{reserve}) = & (\mathit{init}.\mathit{reserve} \rightarrow \mathit{Skip} \fatsemi (\mathit{starts}.\mathit{reserve} \rightarrow \\
& (\mathit{Reserve} \;[\![\, \{\, \mathit{fin}.2 \,\} \,]\!]\; \mathit{fin}.2 \rightarrow \mathit{init}.\mathit{booking} \rightarrow \mathit{Skip}) \\
& [\![\, \{\, \mathit{init}.\mathit{booking} \,\} \,]\!]\; \mathit{init}.\mathit{booking} \rightarrow \mathit{Skip}) \fatsemi P(\mathit{reserve})) \\
& \Box\; (\mathit{fin}.1 \rightarrow \mathit{Skip}) \\
P(\mathit{booking}) = & (\mathit{init}.\mathit{booking} \rightarrow \mathit{Skip} \fatsemi (\mathit{starts}.\mathit{booking} \rightarrow ((\mathit{Booking} \triangle \mathit{init}.\mathit{timeout} \rightarrow \mathit{Stop}) \\
& [\![\, \{\, \mathit{fin}.3, \mathit{fin}.4, \mathit{init}.\mathit{timeout} \,\} \,]\!]\; (\mathit{init}.\mathit{timeout} \rightarrow \mathit{Stop} \\
& \Box\; \mathit{fin}.3 \rightarrow \mathit{init}.\mathit{notify}1 \rightarrow \mathit{Skip} \;\Box\; \mathit{fin}.4 \rightarrow \mathit{init}.\mathit{end} \rightarrow \mathit{Skip})) \\
& [\![\, \{\, \mathit{init}.\mathit{notify}1, \mathit{init}.\mathit{end} \,\} \,]\!]\; (\mathit{init}.\mathit{notify}1 \rightarrow \mathit{Skip} \;\Box\; \mathit{init}.\mathit{end} \rightarrow \mathit{Skip})) \fatsemi \\
& P(\mathit{booking})) \;\Box\; (\mathit{fin}.1 \rightarrow \mathit{Skip}) \\
P(\mathit{timeout}) = & (\mathit{init}.\mathit{timeout} \rightarrow \mathit{Skip} \fatsemi \mathit{starts}.\mathit{timeout} \rightarrow \mathit{Skip} \fatsemi \\
& \mathit{init}.\mathit{notify}2 \rightarrow \mathit{Skip} \fatsemi P(\mathit{notify})) \;\Box\; (\mathit{fin}.1 \rightarrow \mathit{Skip}) \\
P(\mathit{notify}) = & ((\mathit{init}.\mathit{notify}1 \rightarrow \mathit{Skip} \;\Box\; \mathit{init}.\mathit{notify}2 \rightarrow \mathit{Skip}) \fatsemi \\
& \mathit{starts}.\mathit{notify} \rightarrow \mathit{Skip} \fatsemi \mathit{init}.\mathit{abort} \rightarrow \mathit{Skip} \fatsemi P(\mathit{notify})) \;\Box\; (\mathit{fin}.1 \rightarrow \mathit{Skip}) \\
P(\mathit{end}) = & (\mathit{init}.\mathit{end} \rightarrow \mathit{Skip} \fatsemi \mathit{fin}.1 \rightarrow \mathit{Skip}) \\
P(\mathit{abort}) = & (\mathit{init}.\mathit{abort} \rightarrow \mathit{Skip} \fatsemi \mathit{abt}.1 \rightarrow \mathit{Stop}) \;\Box\; (\mathit{fin}.1 \rightarrow \mathit{Skip})
\end{array}
$$

The process *Reserve* describes the semantics of the subprocess *Reservation* upon its syntactic description. We define set J' to index the processes corresponding to the states of the subprocess:

$$
\begin{array}{l}
J' = \{\, \mathit{start}1, \mathit{reseat}, \mathit{end}1 \,\} \\
\mathit{Reserve} = \textbf{let}\; X = \Box\, i : (\alpha Y \setminus \{\, \mathit{fin}.2 \,\}) \bullet (i \rightarrow X \;\Box\; \mathit{fin}.2 \rightarrow \mathit{Skip}) \\
\qquad\qquad\qquad Y = (\; \|\, j : J' \bullet \alpha P(j) \circ P(j)) \\
\qquad\quad \textbf{in}\; (Y \;[\![\, \alpha Y \,]\!]\; X) \setminus \{\!|\mathit{init}|\!\}
\end{array}
$$

where for each j in J', the process $P(j)$ is as defined below; we write m, ranging over $\mathbb{N}$, to denote the number of iterations in the multiple instance *Reserve Seat*:

$$
\begin{array}{l}
P(start1) = (init.rseat \rightarrow Skip \fatsemi fin.2 \rightarrow Skip) \\
P(reseat) = \\
\quad \textbf{let}\ X(n) = ((init.reseat \rightarrow Skip \ \Box\ init.out \rightarrow Skip) \fatsemi \\
\qquad\qquad (n > 1\ \&\ init.in \rightarrow X(n-1) \\
\qquad\qquad \Box\ n = 1\ \&\ init.in \rightarrow init.out \rightarrow init.end1 \rightarrow Skip \\
\qquad\qquad \Box\ init.end1 \rightarrow Skip \ \Box\ n = m\ \&\ init.end1 \rightarrow Skip)) \\
\qquad A(n) = n > 0\ \& \\
\qquad\qquad (init.in \rightarrow Skip \fatsemi starts.reseat \rightarrow Skip \fatsemi init.out \rightarrow Skip \fatsemi A(n-1)) \\
\qquad\qquad \Box\ init.end1 \rightarrow Skip \\
\quad \textbf{in}\ ((X(m) \,[\![\, \{\, init.end1, init.in, init.out \,\} \,]\!]\, A(m)) \fatsemi P(reseat)) \ \Box\ fin.2 \rightarrow Skip \\
P(end1) = (init.end1 \rightarrow Skip \fatsemi fin.2 \rightarrow Skip)
\end{array}
$$

The process *Booking* describes the semantics of the subprocess *Booking* upon its the syntactic description. It is defined as follows, where we define set J'' to index the processes corresponding to the states of the subprocess:

$$
\begin{array}{l}
J'' = \{\, start2, xs3, pbooking, cancel, ticket, end3, end4 \,\} Booking = \\
\quad \textbf{let}\quad X = \Box\, i : (\alpha Y \setminus \{\, fin.3, fin.4 \,\}) \bullet \\
\qquad\qquad\qquad (i \rightarrow X \ \Box\ (fin.3 \rightarrow Skip \ \Box\ fin.4 \rightarrow Skip)) \\
\qquad\quad Y = (\ \|\ j : J'' \bullet \alpha P(j) \circ P(j)) \\
\quad \textbf{in}\ (Y \,[\![\, \alpha Y \,]\!]\, X) \setminus \{\!|init|\!\}
\end{array}
$$

where for each j in J'', the process $P(j)$ is as defined below:

$$
\begin{array}{rl}
P(start2) = & (init.xs3 \rightarrow Skip \fatsemi P(start4)) \ \Box\ (fin.3 \rightarrow Skip \ \Box\ fin.4 \rightarrow Skip) \\
P(xs3) = & (init.xs3 \rightarrow Skip \fatsemi (init.pbooking \rightarrow Skip \ \Box\ init.cancel \rightarrow Skip) \fatsemi P(xs3)) \\
& \Box\ (fin.3 \rightarrow Skip \ \Box\ fin.4 \rightarrow Skip) \\
P(pbooking) = & (init.pbooking \rightarrow Skip \fatsemi starts.pbooking \rightarrow Skip \fatsemi init.ticket \rightarrow Skip \fatsemi \\
& P(pbooking)) \ \Box\ (fin.3 \rightarrow Skip \ \Box\ fin.4 \rightarrow Skip) \\
P(cancel) = & (init.cancel \rightarrow Skip \fatsemi starts.cancel \rightarrow Skip \fatsemi init.end3 \rightarrow Skip \fatsemi \\
& P(cancel)) \ \Box\ (fin.3 \rightarrow Skip \ \Box\ fin.4 \rightarrow Skip) \\
P(ticket) = & (init.ticket \rightarrow Skip \fatsemi starts.ticket \rightarrow Skip \fatsemi init.end4 \rightarrow Skip \fatsemi \\
& P(ticket)) \ \Box\ (fin.3 \rightarrow Skip \ \Box\ fin.4 \rightarrow Skip) \\
P(end3) = & (init.end3 \rightarrow Skip \fatsemi fin.3 \rightarrow Skip) \ \Box\ fin.4 \rightarrow Skip \\
P(end4) = & (init.end4 \rightarrow Skip \fatsemi fin.4 \rightarrow Skip) \ \Box\ fin.3 \rightarrow Skip
\end{array}
$$

5.2 Verifying Consistency of the Airline Reservation System

CSP's behavioural semantics admits refinement orderings under reverse containment, therefore a behavioural specification R can be expressed by constructing the most non-deterministic process satisfying it, called the characteristic process

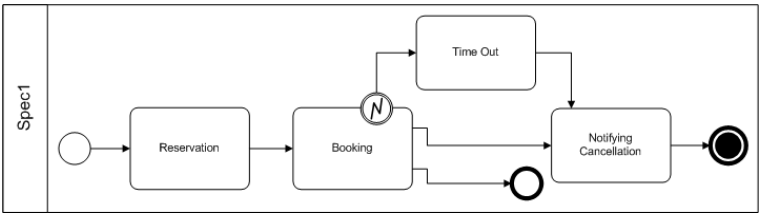


Fig. 3. A BPMN diagram describing the behavioural property defined by process *Spec*1

P_R. Any process Q that satisfies specification R has to refine P_R, denoted by $P_R \sqsubseteq Q$. For example, Figure 3 is a specification of the diagram in Figure 2, abstracting details of subprocesses *Reserve* and *Booking* in the original diagram in Figure 2 into a *task* state.

Letting $K = \{ start3, reserve2, booking2, timeout2, notify2, abort1, end1 \}$, the process *Spec*1 is defined as follows:

$$
\begin{aligned}
Spec1 = \textbf{let}\ & \\
& X = \Box\, i : (\alpha Y \setminus \{fin.1, abt.1\}) \bullet \\
& \qquad (i \rightarrow X \ \Box\ (abt.1 \rightarrow Stop) \ \Box\ (fin.1 \rightarrow Skip)) \\
& Y = \ \|\| \, x : K \bullet \alpha P(x) \circ P(x) \\
\textbf{in}\ & (Y \ [\![\alpha Y]\!] \ X) \setminus \{\!| init |\!\}
\end{aligned}
$$

where for each k in K, the process $P(k)$ is as defined below:

$$
\begin{aligned}
P(start3) &= (init.reserve2 \rightarrow Skip) \fatsemi (fin.1 \rightarrow Skip) \\
P(reserve2) &= ((init.reserve2 \rightarrow Skip) \fatsemi starts.reserve \rightarrow Skip \fatsemi init.booking2 \rightarrow Skip \fatsemi \\
&\quad P(reserve2)) \ \Box\ (fin.1 \rightarrow Skip) \\
P(booking2) &= (init.booking2 \rightarrow Skip \fatsemi starts.booking \rightarrow (Skip \ \triangle\ init.timeout2 \rightarrow Stop) \fatsemi \\
&\quad (init.end1 \rightarrow Skip \ \Box\ init.notify2 \rightarrow Skip) \fatsemi P(booking2)) \ \Box\ (fin.1 \rightarrow Skip) \\
P(timeout2) &= ((init.timeout2 \rightarrow Skip) \fatsemi starts.timeout \rightarrow Skip \fatsemi init.notify3 \rightarrow Skip \fatsemi \\
&\quad P(timeout2)) \ \Box\ (fin.1 \rightarrow Skip) \\
P(notify2) &= ((init.notify2 \rightarrow Skip \ \Box\ init.notify3 \rightarrow Skip) \fatsemi \\
&\quad starts.reserve \rightarrow Skip \fatsemi init.abort1 \rightarrow Skip \fatsemi P(notify2)) \ \Box\ (fin.1 \rightarrow Skip) \\
P(end1) &= (init.end1 \rightarrow Skip \fatsemi fin.1 \rightarrow Skip) \\
P(abort1) &= (init.abort1 \rightarrow Skip \fatsemi abt.1 \rightarrow Stop) \ \Box\ (fin.1 \rightarrow Skip)
\end{aligned}
$$

Note that CSP's traces model is insufficient to verify our models against formal specifications. If we insist on using the traces model, then under traces refinement any process P that has the trace-set $\{ \langle\rangle \}$ will refine and hence satisfy process *Spec*1. Any process which corresponds to a broken or an illegal BPMN diagram might in fact have this trace-set; this demonstrates the inadequacy of the traces model. We therefore use the stable failures model to compare process *Airline* with *Spec*1.

$$Spec1 \sqsubseteq_{\mathcal{F}} Airline \setminus (\alpha Airline \setminus \alpha Spec1)$$

This refinement captures the claim that our semantic model is consistent with respect to different levels of abstraction and *Airline* is indeed a refinement of the abstraction defined by *Spec*1. Due to the specific semantic definition presented in this paper, we are able to verify refinement assertions such as this by model checking using FDR [7].

The above refinement assertion motivates the following generalisation of refinement ordering upon BPMN diagrams. We introduce two types of refinement based on CSP's stable-failures model and the hierarchical composition of BPMN diagrams. We first introduce the notion of hierarchical refinement, where the specification diagram is an abstraction of the implementation diagram via collapsing subprocess states.

Definition 1. ***Hierachical Refinement*** *Given two BPMN diagrams, described by the names n_1 and n_2, and the specification environment l_1 and l_2 respectively, diagram n_1* ***hierachically refines*** *diagram n_2 iff*

$$bsem\ n_2\ l_2 \sqsubseteq_{\mathcal{F}} (bsem\ n_1\ l_1 \setminus S)$$

where S is the set of events corresponding to the alphabet of states that are contained in the subprocess states, which are defined in diagram n_1, and have been abstracted by collapsing them into task states in diagram n_2.

This refinement ordering semantically relates different levels of abstraction between BPMN diagrams. Now we can introduce the notion of hierarchical independence upon behavioural specification.

Definition 2. ***Hierarchical Independence*** *A diagram n_1 in the environment l_1 is a* ***hierarchically independent specification*** *of diagram n_2 in the environment l_2 iff for all names m and specification environments k, the following expression holds:*

$$bsem\ m\ k \sqsubseteq_{\mathcal{F}} (bsem\ n_2\ l_2 \setminus S) \Rightarrow bsem\ n_1\ l_1 \sqsubseteq_{F} bsem\ m\ k$$

where S is the set of events corresponding to the alphabet of states that are contained in the subprocess states, which have been collapsed.

Hierarchical independence allows us to reason about a BPMN diagram against a behavioural specification by verifying a more abstract version of that diagram against the specification. However, sometimes it is not only convenient to hide details of subprocess states, but it is neccessary to also abstract details which are irrelevant to the behavioural property we are interested in.

Definition 3. ***Partial Refinement*** *Given two BPMN diagrams, described by the names n_1 and n_2, and the specification environments l_1 and l_2 respectively, diagram n_1* ***partially refines*** *diagram n_2 iff*

$$bsem\ n_2\ l_2 \sqsubseteq_{\mathcal{F}} (bsem\ n_1\ l_1 \setminus S)$$

where S is the set of event corresponding to the alphabet of all *states that have been abstracted.*

In our example, the diagram in Figure 2 is a partial refinement of the diagram in Figure 3. Conversely we say the diagram in Figure 3 is a partial specification of the diagram in Figure 2. Moreover, these refinement claims may be checked automatically by FDR. These relationships allow a business process developer to focus on the specification of part of the diagram.

6 Conclusion

In this paper, we have presented a process semantics in the language of CSP for a subset of BPMN. We have illustrated by examples how this semantic model may be used to verify that one BPMN diagram is consistent with another, which might be its abstract specification using the same graphical notation. Our semantic model makes it possible to formally analyse and compare BPMN diagrams, and to assert correctness conditions that can be verified using a model checker. Like any development of a complex system, the application of refinement in business process design means that development from an abstract design into an implementation becomes incremental.

The CSP process semantics of a BPMN workflow can be constructed automatically from a simple syntactic presentation of the diagram. We have used Z as a syntactic vehicle, but something like XMI would work too. We do not expect the designer to write in this syntax directly, but to generate it from the diagrammatic notation, annotated with attribute values such as guards and multiplicities.

Future work will include augmenting our semantics with a well-defined transaction and compensation handling, perhaps building on Butler's compensating CSP [3], to provide a formal semantics for the complete BPMN; formalising Property Specification Patterns [6] in CSP, specifically to allow such patterns to be employed for reasoning about behavioural properties of BPMN processes; and automating the semantic translation to facilitate automatic verification.

Acknowledgements

This work is supported by a grant from Microsoft Research. The authors would like to thank anonymous referees for useful suggestions and comments.

References

1. Bolton, C., Davies, J.: Activity graphs and processes. In: Proceedings of the Second International Conference on Integrated Formal Methods, pp. 77–96 (2000)
2. Business Process Execution Language for Web Services, Version 1.1 (May 2003), http://www.ibm.com/developerworks/library/ws-bpel
3. Butler, M., Hoare, T., Ferreira, C.: A trace semantics for long-running transactions. In: Abdallah, A.E., Jones, C.B., Sanders, J.W. (eds.) Communicating Sequential Processes. LNCS, vol. 3525. Springer, Heidelberg (2005)

4. Dijkman, R.M.: Choreography-Based Design of Business Collaborations. BETA Working Paper WP-181, Eindhoven University of Technology (2006)
5. Dijkman, R.M., Dumas, M., Ouyang, C.: Formal semantics and automated analysis of BPMN process models. Technical Report Preprint 5969, Queensland University of Technology (2007)
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Property Specification Patterns for Finite-state Verification. In: 2nd Workshop on Formal Methods in Software Practice (1998)
7. Formal Systems (Europe) Ltd. Failures-Divergences Refinement, FDR2 User Manual (1998), www.fsel.com
8. Foster, H.: Mapping BPEL4WS to FSP. Technical report, Imperial College, London (2003)
9. ILOG JViews BPMN Modeler, http://www.ilog.com/
10. Cámara, J., Canal, C., Cubo, J., Vallecillo, A.: Formalizing WSBPEL Business Processes using Process Algebra. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653. Springer, Heidelberg (2005)
11. Koshkina, M.: Verification of business processes for web services. Master's thesis, York University, Toronto (October 2003)
12. Lucchi, R., Mazzara, M.: A pi-calculus based semantics for WS-BPEL. Journal of Logic and Algebraic Programming 70(1) (January 2007)
13. OMG. Business Process Modeling Notation (BPMN) Specification (February 2006), www.bpmn.org
14. Ouyang, C., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center (2006)
15. Recker, J., Mendling, J.: On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In: Proceedings 18th International Conference on Advanced Information Systems Engineering, pp. 521–532 (2006)
16. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1998)
17. W3C. Web Service Choreography Interface (WSCI) 1.0 (November 2002), http://www.w3.org/TR/wsci
18. Wong, P.Y.H., Gibbons, J.: A Process Semantics for BPMN (extended version) (2007), www.comlab.ox.ac.uk/peter.wong/pub/bpmnsem.pdf
19. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Proof and Refinement. Prentice Hall International Series in Computer Science (1996)

A Formal Descriptive Semantics of UML

Lijun Shan[1] and Hong Zhu[2]

[1] Dept of Computer Science, National Univ. of Defence Tech, Changsha, 410073, China
lijunshan@brookes.ac.uk
[2] Department of Computing, Oxford Brookes University, Oxford OX33 1HX, UK
hzhu@brookes.ac.uk

Abstract. This paper proposes a novel approach to the formal definition of UML semantics. We distinguish descriptive semantics from functional semantics of modelling languages. The former defines which system is an instance of a model while the later defines the basic concepts underlying the models. In this paper, the descriptive semantics of class diagram, interaction diagram and state machine diagram are defined by first order logic formulas. A translation tool is implemented and integrated with the theorem prover SPASS to enable automated reasoning about models. The formalisation and reasoning of models is then applied to model consistency checking.

1 Introduction

With the rapid development of model-driven software development, concerns have been expressed on the semantics of modelling languages such as UML. It is widely recognised that a clear and rigorous semantics of UML is indispensable for rigorous modelling. Unfortunately, in spite of the numerous efforts in the past decade, formal specification of UML has not been satisfactory. This paper proposes a novel approach to defining formal semantics of modelling languages. It is applied to UML class diagram, interaction diagram and state machine diagram. The usefulness of the approach is demonstrated by its implementation in an automated tool and its application to model consistency checking.

The paper is organised as follows. Section 2 describes the proposed approach and discusses the related work. Section 3 elaborates our approach by a formal definition of UML class diagram. Section 4 further discusses how to deal with multiple views defined by separate metamodels and illustrates our approach through defining the semantics of interaction diagram and state machine. Section 5 applies the formal semantics to model consistency checking. Section 6 presents an automated tool, which translates UML models into first order logic and uses a theorem prover SPASS [1] to reason about models. Section 7 concludes the paper and discusses future work.

2 Proposed Approach

2.1 Basic Concepts

As Seidewitz pointed out [2], a software model, like models in any other scientific disciplines, is '*a set of statements about some system under study*', where statements

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 375 – 396, 2008.

are expressions that can be evaluated to a truth value with respect to the modelled systems. Further, Seidewitz stated that a model's meaning has two aspects. One is the model's relationship to the things being modelled. This meaning is implied when saying 'this model means that the Java program must contain these classes'. In this sense, a model is mapped to a collection of systems in a subject domain. By subject domain, we mean a set of systems that a modelling language intends to model, e.g. the collection of Java software systems. Another example of subject domain is the collection of real world systems described with OO concepts. In these subject domains, the truth of a statement like 'a system contains these classes' can be judged.

The other aspect of models' meaning is about the functions and properties of systems being modelled. This meaning is indicated when saying 'an inheritance relation means that every instance of the subclass is also an instance of the superclass'. On this aspect, the meaning of a model is concerned with the basic concepts such as *what is a class*, and their properties and behaviours such as *how the instances of a class behave*. Semantics on this aspect determines the functions of the systems that satisfy a model, and hence whether two models are functionally equivalent even if they look different. To distinguish these two aspects of meanings of models, we call the former *descriptive semantics* and the later *functional semantics*.

From this point of view, we can examine the weakness in the definition of UML semantics. In the UML specification [3], the 'semantics' sections explain properties and structure of each metaclass. Little has been said about how a model is mapped to a collection of systems, or equivalently, how to judge whether a system satisfies a model. Take a simple class diagram that contains one and only one class node labelled with identifier *A* as an example. It can be interpreted in any of the following ways.

- *there is* ***only one*** *class in the system and it is* ***named A***,
- *there is* ***at least one*** *class* ***named A*** *in the system (which may have other classes)*,
- *there is* ***only one*** *class in the system and* ***its name does not matter***,
- *there is* ***at least one*** *class in the system and* ***its name does not matter***.

The official UML documentation does not specify which interpretation of this simplest class diagram is correct. As Kent et al pointed out, a UML model 'typically has more than one possible implementation', and such 'underspecification' must be reflected by explicit definitions of the semantics [4]. However, formalisation of UML descriptive semantics is difficult due to the following reasons.

First, UML is not only for modelling software systems, but also for modelling real world systems, organisations and business processes. Any domain described with OO concepts can be a subject domain. This feature enables UML to bridge the gap between problem domains in the real world and the computation domain, and to model different problem fields including software, hardware, business process, etc. A formal definition of UML semantics must enable such multiple interpretations.

Second, when the full-fledged UML is considered, even the mapping from UML models to systems in a fixed subject domain is non-trivial due to the large set of language elements with complicated interrelations. It is also recognised by many researchers that the official definition of UML contains errors, hence, it evolves rapidly.

Third, UML employs the multiple view principle of modelling. A large number of different types of diagrams can be drawn to model a system from different perspectives. Each type of diagram is defined by one metamodel. These metamodels are

interrelated through references and inheritances between metaclasses. The connections between metamodels further complicate the semantics of the language and also cause a potential serious problem of model inconsistency. A formal definition of UML semantics must be able to deal with such cross references between metamodels.

Another major cause of difficulty comes from the abstraction and underspecification nature of models [4]. UML is intended to be used in different stages of software engineering to describe systems at different levels of abstraction. For example, a model produced at requirements stage should be more abstract than a model built at design stage. The formalisation of UML semantics must reflect the use of the language at different levels of abstraction.

Finally, one of the most important features of UML language is its flexibility. This is achieved by at least two mechanisms. One is the extension mechanism with which new metaclasses can be introduced through the definition of profiles. The other is the under-definition of language elements.

2.2 Related Work

Addressing the underspecification and ambiguity in UML's semantics, remarkable efforts have been made in the past decade to formalise UML semantics. As far as we know, all of them are about the functional semantics or aim at 'a deeper understanding of OO concepts' [5]. The following proposals are among the most well-known.

On the formalisation of class diagram, which is considered the most important type of diagrams in UML, a number of proposals have been advanced. The work by Evans *et al.* defines classifier, association, generalisation and attribute etc. in Z schemas [5]. Relations between objects and classifiers are specified as axioms. Diagrammatical transformation rules are defined as deduction rules to prove properties of UML models. See [6] for a survey of different approaches to formalising class diagram with Z or Object-Z. First order logic (FOL) and description logics (DLs) are used to formalise class diagram [7]. By encoding UML class diagrams in DL knowledge bases, DL reasoning systems can be used to reason about class diagrams. Formalisation of other types of diagrams has also been investigated, especially on state machine diagram. In [8], a rule-based operational semantics of state machine is proposed based on transition systems. Another work on operational semantics of state machine is reported in [9].

Great efforts have been made on formalising different diagrams in one semantic framework. Considering the semantics of a UML model as a set of acceptable structured process, the authors of [10] map class diagrams and state machines into algebraic specifications in Casl-ltl [11]. Another work aiming at integrated semantics of class diagram, object diagram and state machine diagrams is based on graph transformation [12].

To bridge the gap between UML and formal methods, the extensibility mechanism of UML *profile* is used to define specialisations of UML. In [13], a profile UML-B is designed so that the semantics of specialised UML entities is defined via a translation into B. In [14], an integrated formal method combining the process algebra CSP with the specification language Object-Z is used as the intermediate specification language to link UML and Java. A UML profile for CSP-OZ is designed with the aim of generating part of the CSP-OZ specifications from the specialised UML models.

The above existing methods define the semantics of UML by mapping models into a specific semantic domain, such as labelled transition systems, or OO software systems specified in a formal notation such as Z. The properties of OO systems are specified as axioms and used to reason about UML models. In other words, they mostly addressed the functional semantics of UML. Each method focuses on certain properties of OO systems, hence a certain subset of UML is formalised. However, it is hard to see how these approaches could work either alone or together for the full-fledged UML. Most importantly, the ambiguity in descriptive semantics is not addressed in these works. Instead, their semantics formalisations are based on explicit or implicit assumption on the descriptive semantics. Automation of translating UML models to formal specifications to facilitate automated reasoning of UML models has not been achieved in the existing methods.

2.3 Outline of the Proposed Approach

In this paper, we take a novel approach to formalising the semantics of UML models by explicitly distinguishing descriptive semantics from functional semantics and specifying them separately.

First, the descriptive semantics is defined through a mapping from UML models to a set of first order logic statements, which are constructed from a set of predicates and functions via logic connectives and quantifiers. Predicates and functions represent the basic concepts of the modelling language. For example, predicate *Class*(*x*) is defined to represent the concept class in UML. Interrelations between basic concepts as specified in UML metamodels are characterised by a set of axioms, called *axioms of descriptive semantics* in the sequel. The satisfaction of a model by a system is defined as the evaluation of the truth of the statements in the context of the system, provided that how to evaluate these predicates and functions is known.

Second, the functional semantics of UML is defined for the predicates and functions. The properties and dynamic behaviours of modelled systems can be characterised by a set of axioms called *axioms of functional semantics*. Thus, the functional semantics of a model determines the functions and runtime behaviours of the systems that satisfy a model.

Formally, we have the following structure of semantics for a modelling language.

Definition 1. (Semantics of a modelling language) A formal semantic definition of a modelling language consists of the following elements.

- A signature ***Sig***, which defines a formal logic system;
- A set $\boldsymbol{Axm}_D$ of axioms about the descriptive semantics, which is in the formal logic system defined by ***Sig***;
- A set $\boldsymbol{Axm}_F$ of axioms about the functional semantics, which is also in the formal logic systems defined by ***Sig***;
- A mapping ***F*** from models to a set of formulas in the formal logic system defined by ***Sig***. The formulas are the statements for the descriptive semantics of the model;
- A mapping ***H*** from models to a set of formulas in the formal logic system defined by ***Sig***. The formulas represent the hypothesis about the context in which the descriptive semantics is interpreted. □

In the above definition, the signature defines the symbols that can be used in the formulas and axioms. The evaluation of a first order logic formula is as usual.

Definition 2. (Semantics of a model)
Given a semantics definition of a modelling language as in Definition 1, the semantics of a model M under the hypothesis $\boldsymbol{H}$, written $Sem_{\boldsymbol{H}}(M)$, is defined as follows.

$$Sem_{\boldsymbol{H}}(M) = \boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M)$$

where $\boldsymbol{F}(M)$ and $\boldsymbol{H}(M)$ are the sets of statements obtained by applying the semantic mappings $\boldsymbol{F}$ and $\boldsymbol{H}$ to model M, respectively. The descriptive semantics of a model M under the hypothesis $\boldsymbol{H}$, written $DesSem_{\boldsymbol{H}}(M)$, is defined as follows.

$$DesSem_{\boldsymbol{H}}(M) = \boldsymbol{Axm}_D \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M) \quad \square$$

Given a semantics definition of a modelling language in the above framework, reasoning about the properties of a model can be defined as logical inference as follows.

Definition 3. (Properties of a model)
Let $Sem_{\boldsymbol{H}}(M)$ be the semantics of a model M. M has a property P (represented as a formula in the logic system defined by $\boldsymbol{Sig}$) under the semantics definition $Sem_{\boldsymbol{H}}(M)$ and the hypothesis $\boldsymbol{H}$, if and only if $\boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M) \vdash P$ in the formal logic system. Similarly, we say that M has a property P in descriptive semantics, if and only if $\boldsymbol{Axm}_D \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M) \vdash P$ in the formal logic system. $\square$

A key concept of the semantics of modelling languages is the satisfaction of a model by a system. Before defining this concept, let's first define the notion of subject domain and the interpretation of a formal logic in a subject domain.

Definition 4. (Subject domain)
A subject domain $\boldsymbol{Dom}$ of signature $\boldsymbol{Sig}$ with an interpretation Eva is a triple $<D, \boldsymbol{Sig}, Eva>$, where D is a collection of systems on which the formulas of the logic system defined by $\boldsymbol{Sig}$ can be evaluated according to a specific evaluation rule Eva. The value of a formula f evaluated according to the rule Eva in the context of system $s \in D$, written as $Eva(f, s)$, is called the *interpretation* of the formula f in s. We write $s \models_{Eva} f$, if a formula f is evaluated to true in a system $s \in D$, i.e. $s \models_{Eva} f$ iff $Eva(f, s) = true$. $\square$

When there is no risk of confusion, we will omit the subscript in $s \models_{Eva} f$. For a set F of formulas, we write $s \models F$ to denote that for all f in F, $s \models f$.

Definition 5. (Satisfaction of a model)
Let $\boldsymbol{Sig}$ be a given signature and $\boldsymbol{Dom}$ a subject domain of $\boldsymbol{Sig}$. A system s in D *satisfies* a model M according to a semantic definition $Sem_{\boldsymbol{H}}(M)$ if $s \models Sem_{\boldsymbol{H}}(M)$, i.e. for all formulas f in $Sem_{\boldsymbol{H}}(M)$, $s \models f$. $\square$

In the remainder of the paper, we will elaborate the approach by defining the descriptive semantics of UML class diagram, interaction diagram and state machine diagram. We will also demonstrate the application of the semantic definition to model consistency checking.

3 Descriptive Semantics of Class Diagram

3.1 Metamodel

Fig. 1 shows the simplified metamodel on which our formal definition of the descriptive semantics of UML class diagrams is based.

3.2 Derivation of Signature

Given a metamodel, the signature of a formal logic system can be induced by applying the derivation rules defined as follows.

– *Signature Rule* 1: *Unary predicates*. For each metaclass named *MC* in the metamodel, we define a unary atomic predicate *MC*(*x*).
– *Signature Rule* 2: *Binary predicates*. For each association named *MA* between two metaclasses *X* and *Y* in the metamodel, a binary predicates *MA*(*x*, *y*) is defined to represent the relation between elements of type *X* and *Y*.

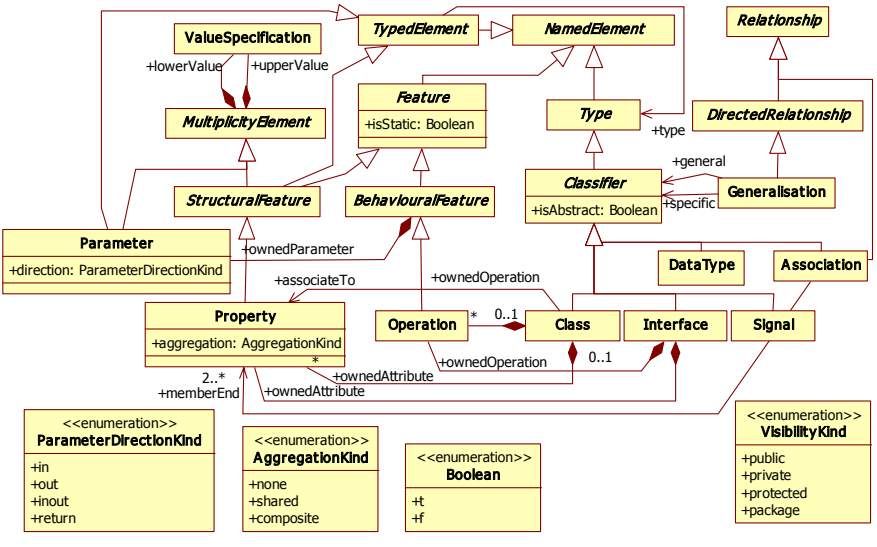


Fig. 1. Metamodel of Class Diagram

A predicate *MC*(*x*) means that element *x* is of type *MC*. For example, a unary predicate *Class*(*x*) is defined to represent the metaclass *Class* in Fig. 1. A binary predicate *MA*(*x*,*y*) means that elements *x* and *y* are related by the relation *MA*. For example, a binary predicate *specific*(*x*, *y*) is defined to represent the association named *specific* from metaclass *Generalisation* to *Classifier* in Fig. 1.

Constants and functions in the signature are also derived from the metamodel.

– *Signature Rule* 3: *Constants*. For each enumeration value *EV* given in an enumeration metaclass *ME* in the metamodel, a constant *EV* is defined.

For example, two enumeration values *t* and *f* are defined in the enumeration metaclass *Boolean* in Fig. 1. Thus, two constants *t* and *f* are defined.

– *Signature Rule* 4: *Functions*. For each meta-attribute *MAttr* of type *MT* in a metaclass *MC*, a function *MAttr* is defined with domain *MC* and range *MT*.

For example, in Fig. 1, metaclass *Classifier* has an attribute *isAbstract* of type *Boolean*. Thus, a function *isAbstract* is defined on domain *Classifier* and range *Boolean*. A statement *isAbstract*(*x*)=*t* means element *x'* property on *isAbstract* is *t*.

Table 2 summarises the constants representing enumeration values and their types, as well as the functions derived from the metamodel in Fig. 1. These functions are partial, i.e. they can be undefined on some elements in a model.

The interpretation of the functions and predicates must be defined in the context of a subject domain. Take the set of C++ programs as an example of subject domain. Given a C++ program, the predicate *Class*(*User*) is true if *User* is a class in the program. The statement *isAbstract*(*User*) is true when the class *User* in the program is declared to be abstract. In this paper, we leave the definition of the interpretation open so that a model can be interpreted in different subject domains.

Table 1. Predicates for Class Diagram

Predicate	Meaning
ValueSpecification(x)	x has type ValueSpecification
MultiplicityElement(x)	x has type MultiplicityElement
StructuralFeature(x)	x has type StructuralFeature
TypedElement(x)	x has type TypedElement
Feature(x)	x has type Feature
BehaviouralFeature(x)	x has type BehaviouralFeature
NamedElement(x)	x has type NamedElement
Type(x)	x has type Type
Classifier(x)	x has type Classifier
Relationship(x)	x has type Relationship
DirectedRelationship(x)	x has type DirectedRelationship
Parameter(x)	x has type Parameter
Property(x)	x has type Property
Operation(x)	x has type Operation
Class(x)	x has type Class
Interface(x)	x has type Interface
Signal(x)	x has type Signal
Generalisation(x)	x has type Generalisation
Association(x)	x has type Association
DataType(x)	x has type DataType
ParameterDirectionKind(x)	x has type ParameterDirectionKind
AggregationKind(x)	x has type AggregationKind
Boolean(x)	x has type Boolean
VisibilityKind(*x*)	x has type VisibilityKind
upperValue(x, y)	the relation between x and y is upperValue
lowerValue(x, y)	the relation between x and y is lowerValue
type(x, y)	the relation between x and y is type
general(x, y)	the relation between x and y is general
specific(x, y)	the relation between x and y is specific
ownedParameter(x, y)	the relation between x and y is ownedParameter
ownedAttribute(x, y)	the relation between x and y is ownedAttribute
ownedOperation(x, y)	the relation between x and y is ownedOperation
associateTo(x, y)	the relation between x and y is AssociateTo
memberEnd(x, y)	the relation between x and y is memberEnd

Table 2. Functions and Constants for Class Diagrams

Function	Domain	Range	
		Type	Values
isStatic	Feature	Boolean	f, t
visibility	NamedElement	VisibilityKind	public, private, protected, package
isAbstract	Classifier	Boolean	f, t
direction	Parameter	ParameterDirectionKind	in, out, inout, return
aggregation	Property	AggregationKind	shared, composite, none

3.3 Axioms

A UML metamodel is a model that defines the abstract syntax of UML diagrams. It can also be regarded as a collection of statements that evaluate to truth values on UML models. A UML model is syntactically valid only if all these statements are true. Thus, they are axioms on the formal systems representing descriptive semantics of models. We identified the following five groups of axioms.

A. Inheritance hierarchy on metaclasses

In a metamodel, concrete metaclasses define types of model elements, while abstract metaclasses define common features of concrete metaclasses. These common features may be specialised by concrete metaclasses. In the sequel, we call a type defined by a concrete metaclass a *concrete type*, and a type defined by an abstract metaclass an *abstract type*. Each element has exactly one concrete type, but may belong to a number of abstract types.

– *Axiom Rule* 1: *Logical implication of inheritance*. For an inheritance relation from metaclass *MA* to *MB*, we have an axiom in the form of $\forall x.\ MA(x) \rightarrow MB(x)$

For example, from the inheritance relation from *Class* to *Classifier* in Fig. 1, an axiom is derived to state that if an element has *Class* as its type, it also belongs to the type *Classifier*. Formally, $\forall x.\ Class(x) \rightarrow Classifier(x)$.

– *Axiom Rule* 2: *Completeness of specialisations*. Let *MA* be a metaclass in a metamodel and MB_1, MB_2, …, MB_k be the set of metaclasses specialising *MA*. We have an axiom in the form of $\forall x.\ MA(x) \rightarrow MB_1(x) \vee MB_2(x) \vee \ldots \vee MB_k(x)$

For example, the following axiom is derived from the metamodel in Fig. 1. It states that if an element has *Classifier* as its type, it must belong to one of the 5 sub-types: *Association*, *DataType*, *Class*, *Interface* or *Signal*.

$\forall x.\ Classifier(x) \rightarrow DataType(x) \vee Association(x) \vee Class(x) \vee Interface(x) \vee Signal(x)$

– *Axiom Rule* 3: *Uniqueness of element classification*. Let MC_1, MC_2, …, MC_n be the set of concrete metaclasses in a metamodel. For each pair of different concrete metaclasses MC_i and MC_j, $i \neq j$, we have an axiom in the following form.

$$\forall x.\ MC_i(x) \rightarrow \neg\, MC_j(x)$$

For example, the following axiom states that if an element has *Property* as its concrete type, it cannot be an *Operation* at the same time.

$$\forall x.\ Property(x) \rightarrow \neg\, Operation(x)$$

B. Navigation between element types

Let *MA* be an association from metaclass MC_1 to MC_2 in a metamodel. For the binary predicate $MA(x,y)$ derived from the association *MA*, the two parameters must be elements of type MC_1 and MC_2, respectively. Thus, we have the following axiom rule.

– *Axiom Rule* 4: *Types of parameters of predicates*. For each binary predicate $MA(x,y)$ derived from an association from metaclass MC_1 to MC_2 in the metamodel, we have an axiom in the following form.

$$\forall x,y.\ MA(x,y) \rightarrow MC_1(x) \wedge MC_2(y)$$

For example, the following axiom is derived from the association *general* from metaclass *Generalisation* to *Classifier* in Fig. 1. It states that if predicate $general(x,y)$ is true, *x* must belong to the type *Generalisation* and *y* must belong to *Classifier*.

$$\forall x,y.\ general(x,y) \rightarrow Generalisation(x) \wedge Classifier(y)$$

Similar to binary predicates, for each function *MAttr*, we have an axiom to specify its domain and range.

– *Axiom Rule* 5: *Domain and range of functions*. For each function *MAttr* derived from a meta-attribute *MAttr* of type *MT* in a metaclass *MC*, we have an axiom in the following form.

$$\forall x,y.\ MC(x) \wedge (MAttr(x) = y) \rightarrow MT(y)$$

For example, for the meta-attribute *isAbstract* of *Classifier* in Fig. 1, the following axiom is derived. It states that if function *isAbstract* is applied on an element of the type *Classifier*, the value of the function must belong to *Boolean*.

$$\forall x,y.\ Classifier(x) \wedge (isAbstract(x) = y) \rightarrow Boolean(y)$$

C. Well-formedness constraints

UML class diagram is insufficient for fully defining the abstract syntax of UML. In complementary, well-formedness constraints are specified in the UML documentation. Some of these well-formedness rules (WFR) are formally defined in OCL, which should also be specified as axioms.

– *Axiom Rule* 6: *Well-formedness rules*. For each WFR formally specified in OCL, we have a corresponding axiom in the first order language.

For example, a WFR in UML document is "*Generalization hierarchies must be directed and acyclical. A classifier cannot be both a transitively general and transitively specific classifier of the same classifier.*" Thus, we have the following axiom.

$$\forall x,\ y.\ Inherit(x,\ y) \rightarrow \neg\ Inherit(y,\ x)$$

where $Inherit(x,y)$ is a binary predicate introduced to simplify the specification of the axiom. It is formally defined by the following two formulas.

$$\forall x,y.\ Generalisation(z) \wedge specific(z,\ x) \wedge general(z,\ y) \rightarrow Inherit(x,\ y)$$

$$\forall x,\ y,\ z.\ Inherit(x,\ y) \wedge Inherit(y,\ z) \rightarrow Inherit(x,\ z)$$

Some well-formedness rules are informally defined in the UML documentation. They cannot be easily specified in first order logic. For example, a rule for *MultiplicityElement* is '*if a non-literal ValueSpecification is used for the lower or upper bound, then*

evaluating that specification must not have side effects'. It cannot be formally specified as an axiom.

D. Definition of enumeration values

We identified three axiom rules to characterise the information contained in each enumeration metaclass.

– *Axiom Rule* 7: *Distinguishability of the literal constants*. For each pair of different literal values a and b defined in an enumeration type, we have an axiom in the form of $a \neq b$.

For example, the metaclass *Boolean* defines two literal values t and f. Thus, we have the axiom $t \neq f$.

– *Axiom Rule* 8: *Type of the literal constants*. For each enumeration value a defined in an enumeration metaclass *ME*, we have an axiom in the form of $ME(a)$ stating that the type of a is *ME*.

For example, for the Boolean values t and f, we have the following two axioms.

$$Boolean(t),\ Boolean(f).$$

– *Axiom Rule* 9: *Completeness of the enumeration*. An enumeration type only contains the listed literal constants as its values, hence for each enumeration metaclass *ME* with literal values $a_1, a_2, \ldots, a_k$, we have an axiom in the form of

$$\forall x.\ ME(x) \text{ -> } (x = a_1) \vee (x = a_2) \vee \ldots \vee (x = a_k)$$

For example, we have the following axiom for the *Boolean* metaclass.

$$\forall x.\ Boolean(x) \text{ -> } (x = t) \vee (x = f)$$

3.4 Translating Models into First Order Logic Formulas

This subsection shows how to translate diagrammatic models to first order logic formulas. We will use the class diagram in Fig. 2 as an example.

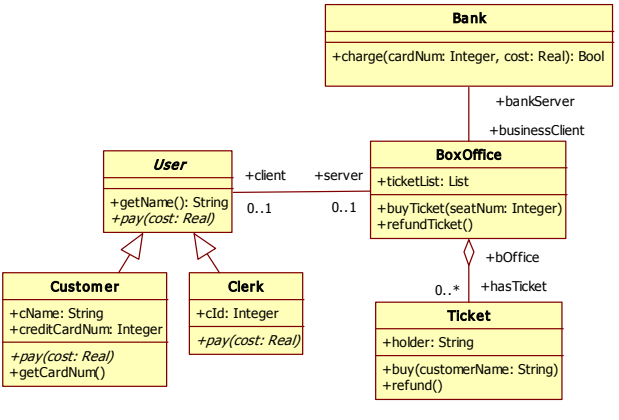


Fig. 2. Class Diagram in the Model Ticketing Office

A. Semantics mapping F_M

For each class diagram, the following rules are applied to generate formulas.

– *Translation Rule* 1: *Classification of elements*. For each identifier *id* of concrete type *MC*, a formula in the form of *MC*(*id*) is generated.

By applying this rule to every element in a diagram, a set of formulas are generated to declare the classification of the identifiers. For example, the following formulas are among those generated from the class nodes in Fig. 2.

$$Class(User),\ Class(Bank),\ Class(BoxOffice).$$

Similarly, formulas are generated by applying other unary predicates that represent concrete types such as *Property*, *Association*, *Operation*, etc.

– *Translation Rule* 2: *Properties of elements*. For each element *a* in the model and every applicable function *MAttr* that represents a meta-attribute, a formula in the form of *MAttr*(*a*)=*v* is generated, where *v* is *a*'s value on the property.

For example, *Clerk* in Fig. 2 is a concrete class. Therefore, the following formula is generated, which states that the value of function *isAbstract* on *Clerk* is false.

$$isAbstract(Clerk) = f$$

Table 3 lists the functions applicable on each concrete type of model elements and the elements contained in the class diagram in Fig. 2. Applicable functions are derived from the metamodel according to the inheritance relation between metaclasses.

– *Translation Rule* 3: *Relationships between elements*. For each related pair (e_1, e_2) of elements in a model, a formula in the form of $R(e_1, e_2)$ is generated to specify the relationship by applying binary predicate $R(x_1, x_2)$.

For example, the class diagram in Fig. 2 depicts a generalisation relation from class *Clerk* to *User*. Hence, we have the formula *specific*(*g*, *Clerk*), where *g* denotes the generalisation arrow.

Table 3. Constants representing model elements

Type	Applicable functions	Elements in Fig.2
Class	isAbstract, visibility	Bank, User, BoxOffice, Clerk, Customer, Ticket
Property	isStatic, aggregation, visibility	name, creditCardNum, ID, ticketList, holder, client, server, businessClient, bankServer, hasTicket, bOffice
Operation	isStatic, visibility	GetName, Pay, GetCardNum, Charge, BuyTicket, Refund-Ticket, Buy, Refund
Association	*isAbstract*, visibility	UserBoxoffice, BoxofficeTicket, BoxofficeBank
DataType	*isAbstract*, *visibility*	String, Integer, Bool, List
Generalisation	/	customerUser, clerkUser
Parameter	direction, visibility	cardNum, cost, seatNum
Signal	isAbstract, visibility	/
Interface	isAbstract, visibility	/
ValueSpecification	/	0, 50, 1, 200

B. Hypothesis mapping H_M

In addition to the above translation rules that are applied to all models, hypothesis rules are needed to generate formulas that represent the meanings of models in specific uses of the modelling language. Their application should be determined by users

according to the situation in which a model is used. The following are some examples of such hypothesis rules.

Let e_1, e_2, …, e_k be the set of elements of a concrete type *MC* in a model.

– *Hypothesis Rule* 1: *Distinguishability of elements*. The hypothesis that the elements of type *MC* in the model are all different can be generated as formulas in the form of $e_i \neq e_j$, for $i \neq j \in \{1,2,\ldots,k\}$.

For example, if it is assumed that in Fig. 2 class *Clerk* is different from class *Customer*, the formula *Clerk* ≠ *Customer* is generated. This hypothesis is applicable if the model is considered as a design, thus force the programmer to implement two classes *Clerk* and *Customer* separately. However, if the model is used as a requirements specification, this hypothesis may not be necessary because a program with one class implementing both *Clerk* and *Customer* can be considered as satisfying the model.

– *Hypothesis Rule* 2: *Completeness of elements*. The hypothesis on the completeness of elements of type *MC* can be generated as a formula in the following form.

$$\forall x.\ MC(x) \rightarrow (x = e_1) \vee (x = e_2) \vee \ldots \vee (x = e_k)$$

For example, the assumption that the model in Fig. 2 contains all classes in the modelled system can be specified as follows.

$$\forall x.\ Class(x) \rightarrow (x = Ticket) \vee (x = Clerk) \vee (x = Customer)$$

$$\vee (x = User) \vee (x = Bank) \vee (x = BoxOffice)$$

This hypothesis on the completeness of classes is applicable when a model represents a system in reverse engineering or as a detailed design. However, when a model is used as requirements specification, an implementation of the system may introduce additional classes and still be regarded as satisfying the requirements. In this case, this hypothesis is not applicable.

Similarly, we have the following hypothesis on the completeness of relations. Let $R(x_1, x_2)$ be a binary predicate, $R(e_{1,1}, e_{1,2})$, $R(e_{2,1}, e_{2,2})$, …, $R(e_{n,1}, e_{n,2})$ be the set of *R* relations contained in the model.

– *Hypothesis Rule* 3: *Completeness of relations*. The hypothesis on the completeness of relation *R* in the model can be generated as a formula in the following form.

$$\forall x_1,x_2.R(x_1,x_2) \rightarrow ((x_1=e_{1,1}) \wedge (x_2=e_{1,2})) \vee ((x_1= e_{2,1}) \wedge (x_2= e_{2,2})) \vee \ldots ((x_1= e_{n,1}) \wedge (x_2= e_{n,2}))$$

This hypothesis assumes that all relations of a certain type are specified in the model, thus any additional relation in a system will be regarded as not satisfying the model. For example, for the model in Fig. 2, we will specify the following formula, if we believe all inheritance relations in the modelled system are depicted in the model.

$$\forall x,y.\ specific(x,y) \rightarrow ((x=ClerkUser) \wedge (y=Clerk)) \vee ((x=CustomerUser) \wedge (y= Customer))$$

It is worth noting that the above hypothesis rules are just examples. They are by no means considered as complete. The point here is the flexibility of UML for different uses can be explicitly revealed through a set of optional hypothesis mappings. How hypothesis rules are related to the use of the modelling language will be an interesting practical problem for further research.

4 Semantics of Interaction and State Machine

Our approach to defining descriptive semantics is applicable on various types of UML diagrams. This section defines the descriptive semantics of interaction diagram and state machine. The same rules and process described in section 3 are applied. The only difference is that their metamodels are connected to the metamodel of class diagram. This section will focus on how to deal with such connections.

4.1 Integration of Metamodels

Fig. 3 shows a simplified metamodel of interaction diagram.

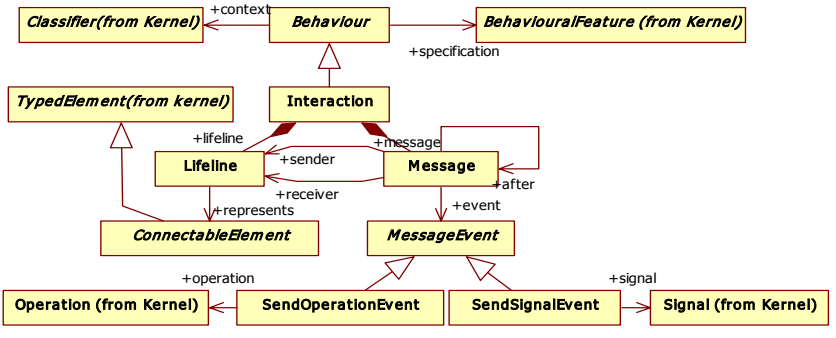


Fig. 3. Metamodel of Interaction Diagram

Metaclasses *Operation*, *Signal*, *TypedElement*, *BehaviouralFeature* and *Classifier* in Fig. 3 were defined in the metamodel of class diagram in Fig. 1 as indicated by 'from Kernel' after their names. They are included in this metamodel to specify the connection between the metamodels. For the associations that relate a metamodel to external metaclasses, the rules for defining predicates and axioms differ from the ordinary rules. For example, in Fig. 3, the association *operation* denotes the correspondence between *SendOperationEvent* in interaction diagram and *Operation* in class diagram. Similarly, the association *signal* denotes the correspondence between *SendSignalEvent* in interaction diagram and *Signal* in class diagram. Thus, the Signature Rule 2 is not applied on them. Instead, such correspondences are specified as axioms about the related element types. The following two axioms are derived from associations *operation* and *signal* in Fig. 3, respectively.

$$\forall x.\ SendOperationEvent(x) \text{ –> } Operation(x)$$

$$\forall x.\ SendSignalEvent(x) \text{ –> } Signal(x)$$

Formally, we have the following general rule for generating axioms from cross metamodel associations.

– *Axiom Rule 10: Cross metamodel association*. For each cross metamodel association from metaclass *MA* to external metaclass *MB*, we have an axiom in the form of $\forall x.\ MA(x) \rightarrow MB(x)$.

Axioms for multiple-view UML models comprise the axioms for different types of diagrams, which are separately derived from the respective metamodels. When the different sets of axioms are integrated, the axioms about '*completeness of specialisations'* have to be modified due to the overlap between the inheritance hierarchies in the different metamodels. Formally,

– *Axiom Rule* 2': *Completeness of specialisations across metamodels*. Let *MA* be a metaclass depicted in two metamodels MM_1 and MM_2. Let metaclasses MB_1, MB_2, …, MB_k be the set of metaclasses that specialise *MA* in metamodel MM_1, and MC_1, MC_2, …, MC_p be the set of metaclasses that specialise *MA* in metamodel MM_2. We have the following axiom when a model is defined by MM_1 and MM_2.

$$\forall x.\ MA(x) \rightarrow MB_1(x) \ \vee \ldots \vee MB_k(x) \vee MC_1(x) \vee \ldots \vee MC_p(x)$$

Take the specialisations of metaclass *TypedElement* in Fig. 1 and Fig. 3 as an example. Axiom (1) below will be derived from Fig. 1 by applying Axiom Rule 2 for defining the semantics of models that only contains class diagrams. Similarly, when a model only contains sequence diagrams, axiom (2) will be used. When the model contains both class diagrams and sequence diagrams, i.e. the models are defined by the two interrelated metamodels, axiom (3) below will be used.

$$\forall x.\ TypedElement(x) \rightarrow Parameter(x) \vee StructuralFeature(x) \tag{1}$$

$$\forall x.\ TypedElement(x) \rightarrow ConnectableElement(x) \tag{2}$$

$$\forall x.\ TypedElement(x) \rightarrow Parameter(x) \vee StructuralFeature(x) \vee ConnectableElement(x) \tag{3}$$

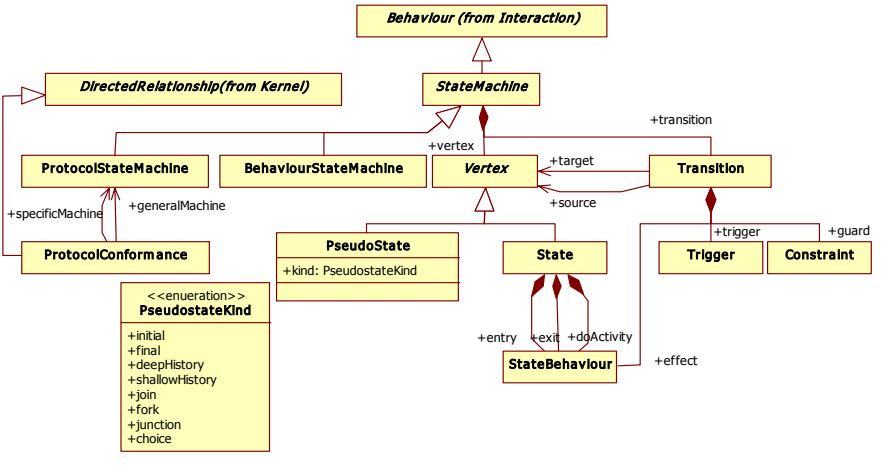


Fig. 4. Metamodel of State Machine Diagram

The signature and axioms of state machine diagrams are derived from the metamodel shown in Fig. 4 by applying the rules given in section 3 and section 4.1. Table 4 summarises the number of generated predicates, functions and axioms.

4.2 Translating Diagrams into First Order Logic Formulas

The translation rules given in section 3 are applied to sequence diagrams and state machines to generate first order logic formulas. For example, the following formulas are among those generated from the interaction diagram shown in Fig. 5 (A).

$$Message(buyTicket)\ ,\ sender(buyTicket,\ c).$$

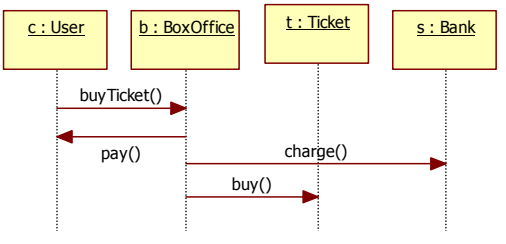
Below are some of the formulas generated from the state machine in Fig. 5 (B).

$State(available)$, $trigger(Transition7,refund)$, $source(Transition7,unavailable)$.

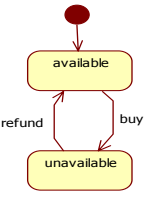
Totally 1459 formulas were generated from the three diagrams of the model Ticketing Office.

Table 4. Summary of the signature and axioms defined for three types of diagrams

			Class Diagram	Interaction Diagram	State Machine
Signature	Unary Predicate	Abstract metaclasses	10	3	2
		Concrete metaclasses	10	5	9
		Enumeration metaclasses	4	0	1
	Binary Predicates		10	8	12
	Functions		5	0	1
	Enumeration constants		13	0	8
Axiom	Inheritance relations		20	4	6
	Completeness of specialisation		10	3	4
	Completeness of classification		10	5	9
	Valid application of binary predicates/functions		15	8	13
	Well-formedness rules		7	1	21



(A) Sequence Diagram



(B) State Machine Diagram

Fig. 5. Sequence Diagram and State Machine Diagram in the Model Ticketing Office

5 Consistency Check: An Application of Descriptive Semantics

The formal definition of UML semantics in our approach naturally facilitates reasoning about models. This section demonstrates the application of the descriptive semantics in consistency checking of models.

Aiming at rigorous modelling, great efforts have been made to define and check models' consistency [15-18], especially in the context of UML models [19-22]. With the definition of model semantics in first order logic, checking the consistency of a model is to prove that the formulas generated from the model are consistent in the context of the axioms. Moreover, additional stronger consistency constraints can also be specified in first order logic. The validity of such constraints, i.e. their consistency with the axioms, can be formally proved.

5.1 Checking Consistency as Logical Inference

Let $\boldsymbol{F}$ be a set of formulas in a signature ***Sig***. As in first order logic, if we can deduce that $\boldsymbol{F}|\!-\!false$, then $\boldsymbol{F}$ is inconsistent. Thus, we have the following definition.

Definition 6. (Logical consistency)
Let $Sem_{\boldsymbol{H}}(M) = \boldsymbol{Axm}_D \cup \boldsymbol{Axm}_F \cup \boldsymbol{F}(M) \cup \boldsymbol{H}(M)$ be the semantics of a model M. Model M is said to be *logically inconsistent* in the semantic definition $Sem_{\boldsymbol{H}}(M)$ if $Sem_{\boldsymbol{H}}(M)|\!-\!false$; otherwise, we say that the model is *logically consistent*. □

It is easy to see that a logically inconsistent model is not satsifiable in a subject domain whose interpretation of formulas is consistent with the logic system.

Definition 7. (Consistent interpretation of formulas in a subject domain)
Let ***Dom***=<D, ***Sig***, Eva> be a subject domain as defined in Definition 4. The interpretation of formulas in signature ***Sig*** is consistent with first order logic if and only if for all formulas q and $p_1, p_2, \ldots, p_k$ that $p_1, p_2, \ldots, p_k \vdash q$, and for all systems s in D that $Eva(p_i, s) = true$ for i=1,2,…, k, we always have $Eva(q, s) = true$. □

Theorem 1. (Unsatisfiability of inconsistent model)
A model M that is logically inconsistent in the semantic definition $Sem_{\boldsymbol{H}}(M)$ is not satisfiable on any subject domain whose interpretation of formulas is consistent with first order logic.

Proof. We prove by contradiction. Let M be a logically inconsistent model, s be a system in a subject domain ***Dom*** that satisfies the model according to the semantic definition $Sem_{\boldsymbol{H}}(M)$. By Definition 5, for all formulas p in $Sem_{\boldsymbol{H}}(M)$, $s|\!=\!p$. By Definition 6, M is logically inconsistent means that $Sem_{\boldsymbol{H}}(M)|\!-\!false$. By the property that the interpretation of formulas in the subject domain ***Dom*** is consistent with the first order logic, it follows (Definition 8) that $s|\!=\!false$. Thus, we find a contradiction. Therefore, the theorem is true. □

In the experiment, we used SPASS theorem prover to prove that each set of the formulas generated from the three diagrams in the model Ticketing Office shown in Fig. 2 and Fig. 5 are logically consistent. Their union is also consistent. Moreover, the set of axioms for class diagrams, interaction diagrams and state machines are also proven to be logically consistent. Thus, we have the following theorem.

Theorem 2. (Consistency of the axioms in semantics definition)
The sets of axioms generated from the metamodels for class diagrams, interaction diagrams and state machines are consistent as they are individually as well as together.

Proof. As stated above. □

We have also made various minor changes to the diagrams in the model Ticketing Office to demonstrate that some changes can lead to logically inconsistent set of formulas, thus proved the existence of inconsistent models in UML according to our semantic definition. Thus, it is feasible to check models' consistency through logic inferences based on descriptive semantics.

It is worth noting that, generally speaking, logical consistency does not guarantee that the model is satsifiable in a subject domain.

5.2 Checking Consistency against Additional Constraints

In addition to checking the consistency of a model as described in the previous subsection, it is often desirable to check models against addition constraints. For example, the following consistency constraint has been studied in the literature [23, 24]. It states that a life line must represent an instance of a class.

$$\forall x, y, z.\ Lifeline(x) \wedge represent(x,y) \wedge type(y, z) \rightarrow Class(z)$$

If a consistency constraint cannot be derived from the axioms, a model that is logically consistent does not necessarily satisfy the additional constraint. Thus, we have the following notion of consistency with respect to a set of constraints.

Definition 8. (Consistency w.r.t. consistency constraints)
Given a set of consistency constraints $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$, the consistency of a model M with respect to the constraints $\boldsymbol{C}$ under the semantics definition $Sem_{\boldsymbol{H}}(M)$ is the consistency of the set $\boldsymbol{U} = Sem_{\boldsymbol{H}}(M) \cup \boldsymbol{C}$ of formulas. In particular, we say that a model fails on a specific constraint c_k, if $Sem_{\boldsymbol{H}}(M)$ is consistent, but $Sem_{\boldsymbol{H}}(M) \cup \{c_k\}$ is not. □

The following are some commonly used consistency constraints.

- Message represents operation call of the message receiver [23]. Formally,
$$\forall x, y, z, u.\ Message(x) \wedge event(x,y) \wedge SendOperationCall(y) \wedge receiver(x,z) \wedge type(z, u) \rightarrow ownedOperation(u,y)$$
- The classifier of a message's sender must be associated to the classifier of the message's receiver [23]. Formally,
$$\forall x,y,z,u,v.\ Message(x) \wedge sender(x,y) \wedge type(y,u) \wedge receiver(x,z) \wedge type(z,v) \rightarrow \exists\, w,m,n.\ Association(w) \wedge memberEnd(w, m) \wedge memberEnd(w, n) \wedge AssociateTo(m, u) \wedge AssociateTo(n,v)$$
- Protocol transition refers to an operation (i.e., has a call trigger corresponding to an operation), and that operation applies to the context classifier of the state machine of the protocol transition. Formally,
$$\forall x,y,z.\ ProtocolStateMachine(x) \wedge transition(x,y) \wedge trigger(y,z) \wedge context(x,u) \rightarrow Operation(z) \wedge ownedOperation(u,z)$$
- The order of messages in interaction diagram must be consistent with the order of triggers on transitions in state machine diagram [23, 25]

$$\forall x,y,z,u.Message(x) \wedge event(x,z) \wedge Message(y) \wedge event(y,u) \wedge after(x,y) \rightarrow Trigs(z,u)$$

where $Trigs(x,y)$ is an auxiliary predicate defined as follows.

$$\forall x,y,z,u,v.\ Transition(x) \wedge trigger(x,u) \wedge target(x,y) \wedge Transition(z) \wedge trigger(z,v) \wedge source(z,y) \rightarrow Trigs(v,u)$$

$$\forall x,y,z.\ Trigs\ (x,y) \wedge Trigs\ (y,z) \rightarrow Trigs\ (x,z)$$

In the above discussion, we have made an implicit assumption about the validity of the constraints. Informally, a constraint is invalid if it conflicts with the semantics axioms of the language and thus cannot be satisfied by any model. Here, we distinguish two types of validity: descriptive validity and functional validity.

Definition 9. (Validity of consistency constraints)
Let $\boldsymbol{A}_D$ and $\boldsymbol{A}_F$ be the sets of axioms for descriptive semantics and functional semantics, respectively. A set $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$ of consistency constraints is *descriptively valid* if $\boldsymbol{A}_D \cup \boldsymbol{C}$ is logically consistent. The set $\boldsymbol{C}$ of consistency constraints is functionally valid $\boldsymbol{A}_D \cup \boldsymbol{A}_F \cup \boldsymbol{C}$ is logically consistent. □

We have conducted an experiment with the validity of consistency constraints using SPASS. It is proved that the constraints given above are all descriptively valid.

A consistency constraint can be ineffective if it does not impose any additional restriction on models. This is true if the constraint can be deduced from the axioms in first order logic. Thus, we have the following definition.

Definition 10. (Effectiveness of consistency constraints)
Let $\boldsymbol{A}$ be a set of semantics axioms. A set $\boldsymbol{C}=\{c_1, c_2, \ldots, c_n\}$ of consistency constraints is *logically ineffective* with respect to the set $\boldsymbol{A}$ of axioms if $\boldsymbol{A} \vdash \boldsymbol{C}$. □

Obviously, if $\boldsymbol{C}$ is logically ineffective, a model logically consistent in the context of axiom $\boldsymbol{A}$ will be consistent with respect to $\boldsymbol{C}$.

The consistency constraints given above are all proven to be not ineffective.

6 Implementation of Semantics Translation Tool

By translating UML models into first order logic statements, reasoning about models can be realised as logical inferences and automated by using a theorem prover. We have designed and implemented a tool Translator to translate UML models to first order logic statements. The tool is integrated with a modelling tool and a theorem prover. Fig. 6 shows the structure and workflow of the tools.

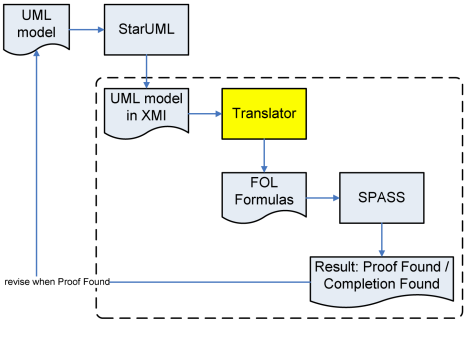


Fig. 6. Process of formalising and reasoning UML models

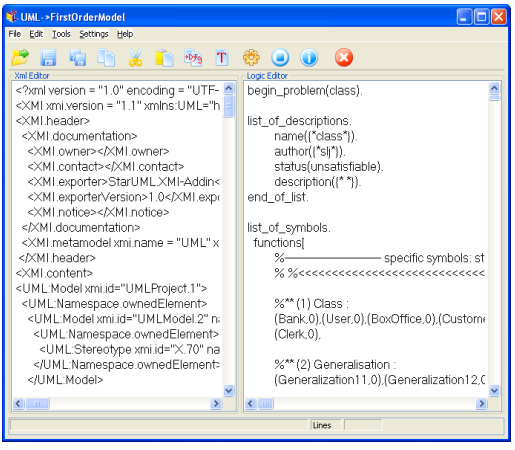


Fig. 7. Snapshot of Translator

The input to Translator is UML models in XMI formats. StarUML [26], a UML modelling tool, is used to generate XMI representation of UML models. The output of our tool is a text file that is readable by SPASS, which is an automated theorem prover for first order logic with equality. Fig. 7 gives a screen snapshot of Translator, where *XMI editor* on the left displays the input XMI file and *Logic editor* on the right displays the generated formulas in SPASS input format.

When SPASS is invoked with an input generated by Translator, the consistency of the statements is inferred. In particular, it infers whether $\boldsymbol{S} \vdash \mathit{false}$ can be proved, where $\boldsymbol{S}$ is the set of formulas including the axioms, hypothesis and formulas generated from the model and optionally some consistency constraints. Since SPASS is refutationally complete [27], if the set of statements in $\boldsymbol{S}$ is logically inconsistent, the system terminates with '*proof found*' and outputs a proof of false; otherwise if it terminates with '*completion found*', which means no proof of false can be found, so $\boldsymbol{S}$ is logically consistent.

We have used the tool to conduct a number of experiments on reasoning about interesting properties of UML diagrams. These experiments include checking the consistency of the axioms, checking model consistency without additional constraints and with various additional constraints, checking consistency constraints' validity and effectiveness, etc. Details of the experiments will be reported separately.

7 Conclusion

The main contribution of this paper is three-fold. First, we introduced the notions of descriptive semantics and functional semantics, and proposed a general framework for separately defining these two aspects of semantics of modelling languages. Second, we proposed a systematic technique to formally specify the descriptive semantics of UML in first order logic, which include the rules for rigorously inducing first order languages from metamodels, the rules for systematically deriving axioms from metamodels, and the rules for automatic translating models into formulas. Third, we

successfully applied the technique to UML class diagram, interaction diagram and state machine. We also demonstrated the usefulness of the formal definition of descriptive semantics by applying it to model consistency checking, and thus laid a logic foundation for consistency checking.

Our approach has the following distinctive features in comparison with existing methods, which are in complementary to ours in the sense that they mostly defined the functional aspect of semantics.

First, our approach explicitly separates descriptive semantics from functional semantics of modelling languages. This enables the definition of the descriptive aspect of semantics to be abstract in the sense that it is independent of any subject domain. This reflects the practical uses of UML that a same model describes both real world systems and computer information systems.

Second, by introducing the notion of hypothesis in semantic definition, our approach achieves the flexibility of semantics of UML models, i.e. the same language is used for various purposes in software development.

Third, the approach is practically useful as we demonstrated the successful application of the approach on non-trivial subsets of class diagrams, interaction diagrams and state machines. In particular, our approach provides a natural and nice solution to the problem in defining multiple view modelling languages where each view is defined by one metamodel and these meat-models are interconnected.

Moreover, the translation from UML models to semantics can be rigorously defined. The translation for the subset of class diagram, interaction diagram and state machine has been implemented and tested.

Finally, the semantic definition facilitates formal and automated reasoning about models. We have demonstrated the application of such reasoning to a well-known non-trivial problem of software modelling, i.e. consistency checking. Experiments have shown promising results.

We are further researching on the definition of the functional semantics of UML in a form that can be nicely linked to descriptive semantics reported in this paper. We are also investigating logical properties of the semantic definitions.

In our investigation of UML semantics, we found a number of errors in its metamodel. Some of them were corrected in the simplified metamodel presented in this paper. More details will be reported separately.

Acknowledgement

The work reported in this paper was done during Lijun Shan's visit at the Oxford Brookes University, which is funded by China Scholarship Council. The project is partly funded by the China Ministry of Science and Technology in the National Basic Research Program (973 program) under the grant 2005CB321800 and the Foundation of Postgraduate Innovation at National University of Defined Technology, China. The authors are grateful to the members of the Applied Formal Methods research group at the school of technology of Oxford Brookes University, especially Dr. Ian Bayley and Mr Lamine Ouyahia, for numerous discussions on related topics.

References

1. SPASS, http://spass.mpi-inf.mpg.de/
2. Seidewitz, E.: What models mean. IEEE Software 20(5), 26–31 (2003)
3. OMG, Unified Modeling Language: Superstructure version 2.0. Object Management Group (2005)
4. Kent, S., Evans, A., Rumpe, B.: UML Semantics FAQ. In: Moreira, A.M.D., Demeyer, S. (eds.) ECOOP 1999 Workshops. LNCS, vol. 1743, pp. 33–56. Springer, Heidelberg (1999)
5. Evans, A., et al.: The UML as a Formal Modeling Notation. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 325–334. Springer, Heidelberg (1999)
6. Amálio, N., Polack, F.: Comparison of Formalisation Approaches of UML Class Constructs in Z and Object-Z. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 339–358. Springer, Heidelberg (2003)
7. Berardi, D., Cal, A., Calvanese, D.: Reasoning on UML class diagrams Artificial Intelligence 168(1), 70–118 (2005)
8. Varro, D.: A Formal Semantics of UML Statecharts by Model Transition Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 378–392. Springer, Heidelberg (2002)
9. Beeck, M.v.d.: A structured operational semantics for UML-statecharts. Softw. Syst. Model 1, 130–141 (2002)
10. Reggio, G., Cerioli, M., Astesiano, E.: Towards a Rigorous Semantics of UML Supporting Its Multiview Approach. In: Hussmann, H. (ed.) FASE 2001. LNCS, vol. 2029, pp. 171–186. Springer, Heidelberg (2001)
11. Reggio, G., Astesiano, E., Choppy, C.: Casl-Ltl : A Casl Extension for Dynamic Reactive Systems – Summary. Technical Report DISI-TR-99-34. DISI – Universit‘a di Genova, Italy (1999)
12. Kuske, S., et al.: An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 11–28. Springer, Heidelberg (2002)
13. Snook, C., Butler, M.: UML-B: Formal Modeling and Design Aided by UML. ACM Transactions on Software Engineering and Methodology 15(1), 92–122 (2006)
14. Moller, M., et al.: Linking CSP-OZ with UML and Java: A Case Study. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 267–286. Springer, Heidelberg (2004)
15. Nentwich, C., et al.: Flexible consistency checking. ACM Transactions on Software Engineering and Methodology 12(1), 28–63 (2003)
16. Nentwich, C., et al.: xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Techn. 2(2), 51–185 (2002)
17. Shan, L., Zhu, H.: Specifying consistency constraints for modelling languages. In: 18th International Conference on Software Engineering and Knowledge Engineering (SEKE 2006), pp. 578–583. Knowledge Systems Institute, San Francisco (2006)
18. Shan, L., Zhu, H.: Consistency check in modeling multi-agent systems. In: 26th International Computer Software and Applications Conference (COMPSAC 2004), pp. 114–121. IEEE Computer Society, Hong Kong (2004)
19. Muskens, J., Bril, R.J., Chaudron, M.R.V.: Generalizing Consistency Checking between Software Views. In: 5th Working IEEE/IFIP Conference on Software Architecture (WICSA 2005), pp. 169–180. IEEE Computer Society, Los Alamitos (2005)

20. Rasch, H., Wehrheim, H.: Cheking Consistency in UML Diagrams: Classes and State Machines. In: Formal Methods for Open Object-Based Distributed Systems, pp. 229–243. Springer, Heidelberg (2003)
21. Simmonds, J., Bastarrica, M.C.: A Tool for Automatic UML Model Consistency Checking. In: ASE 2005, pp. 431–432. ACM, Long Beach (2005)
22. Straeten, R.V.D., et al.: Using Description Logic to Maintain Consistency between UML Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
23. Egyed, A.: Instant Consistency Checking for the UML. In: ICSE 2006, Shanghai, China, pp. 381–390 (2006)
24. Straeten, R.V.D., Simmonds, J., Mens, T.: Detecting Inconsistencies between UML Models Using Description Logic. In: Proceedings of the 2003 International Workshop on Description Logics (DL 2003), Rome, Italy (2003)
25. Mens, T., Straeten, R.V.D., Simmonds, J.: Maintaining Consistency between UML Models with Description Logic Tools. In: ECOOP Workshop on Object-Oriented Reengineering (2003)
26. StarUML, `http://staruml.sourceforge.net/en/`
27. Weidenbach, C.: SPASS - Version 0.49. J. Autom. Reasoning 18(2), 247–252 (1997)

Author Index