Jin Song Dong
Huibiao Zhu (Eds.)

LNCS 6447

Formal Methods and Software Engineering

12th International Conference on Formal Engineering Methods, ICFEM 2010
Shanghai, China, November 2010, Proceedings

Lecture Notes in Computer Science 6447

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

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Jin Song Dong Huibiao Zhu (Eds.)

Formal Methods and Software Engineering

12th International Conference
on Formal Engineering Methods, ICFEM 2010
Shanghai, China, November 17-19, 2010
Proceedings

Volume Editors

Jin Song Dong
National University of Singapore
School of Computing, Computer Science Dept.
13 Computing Drive, Singapore 117417, Singapore
E-mail: dongjs@comp.nus.edu.sg

Huibiao Zhu
East China Normal University
Software Engineering Institute
3663 Zhongshan Road (North), Shanghai, 200062, China
E-mail: hbzhu@sei.ecnu.edu.cn

Library of Congress Control Number: 2010938033

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

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-642-16900-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-16900-7 Springer Berlin Heidelberg New York

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

springer.com

© Springer-Verlag Berlin Heidelberg 2010
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

Formal methods have made significant progress in recent years with successful stories from Microsoft (SLAM project), Intel (i7 processor verification) and NICTA/OK-Lab (formal verification of an OS kernel). The main focus of formal engineering methods lies in how formal methods can be effectively integrated into mainstream software engineering. Various advanced theories, techniques and tools have been proposed, developed and applied in the specification, design and verification of software or in the construction of software. The challenge now is how to integrate them into engineering development processes to effectively deal with large-scale and complex computer systems for their correct and efficient construction and maintenance. This requires us to improve the state of the art by researching effective approaches and techniques for integration of formal methods into industrial engineering practice, including new and emerging practice.

This series, International Conferences on Formal Engineering Methods, brings together those interested in the application of formal engineering methods to computer systems. This volume contains the papers presented at ICFEM 2010, the 12th International Conference on Formal Engineering Methods, held November 17–19, in Shanghai, China, in conjunction with the Third International Symposium on Unifying Theories of Programming (UTP 2010).

The Program Committee received 114 submissions from 29 countries and regions. Each paper was reviewed by at least three program committee members. After extensive discussion, the Program Committee decided to accept 42 papers for presentation, leaving many good-quality papers behind. We owe a great deal to the members of Program Committee and external reviewers. The program also included three invited talks by Matthew Dwyer from the University of Nebraska-Lincoln, Kokichi Futatsugi from Japan Advanced Institute of Science and Technology and Wang Yi from Uppsala University.

ICFEM 2010 was organized by the Software Engineering Institute, East China Normal University. We would like to express our sincere thanks to the staff members and students for their organizational assistance, in particular Geguang Pu, Jian Guo, Min Zhang, Qin Li and Mengying Wang. 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.

November 2010

Jin Song Dong
Huibiao Zhu

Conference Organization

Steering Committee

Keijiro Araki
Jin Song Dong
Chris George
Jifeng He
Mike Hinchey
Shaoying Liu (Chair)
John McDermid
Tetsuo Tamai
Jim Woodcock

Conference Chair

Jifeng He

Program Chairs

Jin Song Dong
Huibiao Zhu

Program Committee

Yamine Ait Ameur
Nazareno Aguirre
Bernhard Aichernig
Keijiro Araki
Farhad Arbab
Richard Banach
Jonathan Bowen
Karin Breitman
Michael Butler
Andrew Butterfield
Ana Cavalcanti
Chunqing Chen
Mingsong Chen
Wei-Ngan Chin
Jim Davies
Zhenghua Duan
Colin Fidge
John Fitzgerald
Joaquim Gabarro
Stefania Gnesi
Mike Hinchey
Thierry Jeron
Gerwin Klein
Kim Larsen
Michael Leuschel
Xuandong Li
Zhiming Liu
Shaoying Liu
Brendan Mahony
Tom Maibaum
Tiziana Margaria
Dominique Mery
Huaikou Miao
Flemming Nielson

Jun Pang
Geguang Pu
Shengchao Qin
Zongyan Qiu
Anders P. Ravn
Augusto Sampaio
Marjan Sirjani
Graeme Smith
Jing Sun
Jun Sun
Kenji Taguchi
Yih-Kuen Tsay
T.H. Tse
Sergiy Vilkomir
Xu Wang
Ji Wang
Hai Wang
Heike Wehrheim
Jim Woodcock
Wang Yi
Jian Zhang

Local Organization

Jian Guo, Qin Li, Geguang Pu (Chair), Min Zhang

Webmaster

Mengying Wang

External Reviewers

Nuno Amalio
June Andronick
Thomas Bøgholm
Granville Barnett
Lei Bu
Josep Carmona
Valentin Cassano
Pablo Castro
Shengbo Chen
Yu-Fang Chen
Zhenbang Chen
Robert Clarisó
John Colley
Phan Cong-Vinh
Marcio Cornelio
Andreea Costea
Florin Craciun
Kriangsak Damchoom
Jordi Delgado
Yuxin Deng
Brijesh Dongol
Andrew Edmunds
Alessandro Fantechi
Gianluigi Ferrari
Marc Fontaine
Cristian Gherghina
Paul Gibson
Jian Guo
Henri Hansen
Ian J. Hayes
Guanhua He
Elisabeth Jöbstl
Swen Jacobs
Mohammad Mahdi Jaghoori
Ryszard Janicki
Li Jiao
Jorge Julvez
Narges Khakpour
Ramtin Khosravi
Rafal Kolanski
Willibald Krenn
Shigeru Kusakabe
Bixin Li
Jianwen Li

Qin Li
Xiaoshan Li
Sheng Liu
Michele Loreti
Chenguang Luo
Mingsong Lv
Abdul Rahman Mat
Franco Mazzanti
Lijun Mei
Hiroshi Mochio
Charles Morisset
Alexandre Mota
Toby Murray
Benaissa Nazim
Sidney Nogueira
Ulrik Nyman
Kazuhiro Ogata
Joseph Okika
Yoichi Omori
Fernando Orejas
David Parker
Richard Payne
German Regis
Hideki Sakurada
Martin Schäf
Thomas Sewell
K.C. Shashidhar
Neeraj Singh
Jiri Srba
Kohei Suenaga
Tian Huat Tan
Claus Thrane
Ming-Hsien Tsai
Saleem Vighio
Shuling Wang
Xi Wang
Zheng Wang
Liu Wanwei
Kirsten Winter
Tobias Wrigstad
Zhongxing Xu
Shaofa Yang
Yu Yang
Fang Yu
Hongwei Zeng
Chenyi Zhang
Shaojie Zhang
Xian Zhang
Jianhua Zhao
Liang Zhao
Manchun Zheng

Table of Contents

Invited Talks

Fostering Proof Scores in CafeOBJ 1
Kokichi Futatsugi

Exploiting Partial Success in Applying Automated Formal Methods (Abstract) 21
Matthew B. Dwyer

Multicore Embedded Systems: The Timing Problem and Possible Solutions (Abstract) 22
Wang Yi

Theorem Proving and Decision Procedures

Applying PVS Background Theories and Proof Strategies in Invariant Based Programming 24
Johannes Eriksson and Ralph-Johan Back

Proof Obligation Generation and Discharging for Recursive Definitions in VDM 40
Augusto Ribeiro and Peter Gorm Larsen

Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq 56
Iman Poernomo and Jeffrey Terrell

Decision Procedures for the Temporal Verification of Concurrent Lists 74
Alejandro Sánchez and César Sánchez

An Improved Decision Procedure for Propositional Projection Temporal Logic 90
Zhenhua Duan and Cong Tian

Web Services and Workflow

A Semantic Model for Service Composition with Coordination Time Delays 106
Natallia Kokash, Behnaz Changizi, and Farhad Arbab

Compensable WorkFlow Nets 122
Fazle Rabbi, Hao Wang, and Wendy MacCaull

Automatically Testing Web Services Choreography with Assertions 138
Lei Zhou, Jing Ping, Hao Xiao, Zheng Wang, Geguang Pu, and Zuohua Ding

Applying Ordinary Differential Equations to the Performance Analysis of Service Composition 155
Zuohua Ding, Hui Shen, and Jing Liu

Verification I

Verifying Heap-Manipulating Programs with Unknown Procedure Calls 171
Shengchao Qin, Chenguang Luo, Guanhua He, Florin Craciun, and Wei-Ngan Chin

API Conformance Verification for Java Programs 188
Xin Li, H. James Hoover, and Piotr Rudnicki

Assume-Guarantee Reasoning with Local Specifications 204
Alessio Lomuscio, Ben Strulo, Nigel Walker, and Peng Wu

Automating Coinduction with Case Analysis 220
Eugen-Ioan Goriac, Dorel Lucanu, and Grigore Roşu

Applications of Formal Methods

Enhanced Semantic Access to Formal Software Models 237
Hai H. Wang, Danica Damljanovic, and Jing Sun

Making Pattern- and Model-Based Software Development More Rigorous 253
Denis Hatebur and Maritta Heisel

Practical Parameterised Session Types 270
Andi Bejleri

A Formal Verification Study on the Rotterdam Storm Surge Barrier 287
Ken Madlener, Sjaak Smetsers, and Marko van Eekelen

Verification II

Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems 303
José Meseguer and Peter Csaba Ölveczky

Automated Multiparameterised Verification by Cut-Offs 321
Antti Siirtola

Automating Cut-off for Multi-parameterized Systems 338
Youssef Hanna, David Samuelson, Samik Basu, and Hridesh Rajan

Method for Formal Verification of Soft-Error Tolerance Mechanisms in Pipelined Microprocessors 355
Miroslav N. Velev and Ping Gao

Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP 371
Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne

Probability and Concurrency

Model Checking Hierarchical Probabilistic Systems 388
Jun Sun, Songzheng Song, and Yang Liu

Trace-Driven Verification of Multithreaded Programs 404
Zijiang Yang and Karem Sakallah

Closed Form Approximations for Steady State Probabilities of a Controlled Fork-Join Network 420
Jonathan Billington and Guy Edward Gallasch

Reasoning about Safety and Progress Using Contracts 436
Imene Ben-Hafaiedh, Susanne Graf, and Sophie Quinton

Program Analysis

Abstract Program Slicing: From Theory towards an Implementation 452
Isabella Mastroeni and Đurica Nikolić

Loop Invariant Synthesis in a Combined Domain 468
Shengchao Qin, Guanhua He, Chenguang Luo, and Wei-Ngan Chin

Software Metrics in Static Program Analysis 485
Andreas Vogelsang, Ansgar Fehnker, Ralf Huuck, and Wolfgang Reif

A Combination of Forward and Backward Reachability Analysis Methods 501
Kazuhiro Ogata and Kokichi Futatsugi

Model Checking

Model Checking a Model Checker: A Code Contract Combined Approach 518
Jun Sun, Yang Liu, and Bin Cheng

On Symmetries and Spotlights – Verifying Parameterised Systems 534
Nils Timm and Heike Wehrheim

A Methodology for Automatic Diagnosability Analysis 549
Jonathan Ezekiel and Alessio Lomuscio

Making the Right Cut in Model Checking Data-Intensive Timed Systems 565
Rüdiger Ehlers, Michael Gerke, and Hans-Jörg Peter

Comparison of Model Checking Tools for Information Systems 581
Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar

Object Orientation and Model Driven Engineering

A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model 597
Scott West, Sebastian Nanz, and Bertrand Meyer

Model-Driven Protocol Design Based on Component Oriented Modeling 613
Prabhu Shankar Kaliappan, Hartmut König, and Sebastian Schmerl

Laws of Pattern Composition 630
Hong Zhu and Ian Bayley

Dynamic Resource Reallocation between Deployment Components 646
Einar Broch Johnsen, Olaf Owe, Rudolf Schlatte, and Silvia Lizeth Tapia Tarifa

Specification and Verification

A Pattern System to Support Refining Informal Ideas into Formal Expressions 662
Xi Wang, Shaoying Liu, and Huaikou Miao

Specification Translation of State Machines from Equational Theories into Rewrite Theories 678
Min Zhang, Kazuhiro Ogata, and Masaki Nakamura

Alternating Interval Based Temporal Logics 694
Cong Tian and Zhenhua Duan

Author Index 711

Fostering Proof Scores in CafeOBJ

Kokichi Futatsugi

Graduate School of Information Science &
Research Center for Software Verification,
Japan Advanced Institute of Science and Technology (JAIST),
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
futatsugi@jaist.ac.jp

Abstract. *Proof scores* are instructions to a proof engine such that when executed, if everything evaluates as expected, then a desired theorem is proved. Proof scores hide the detailed calculations done by machines, while revealing the proof plan created by humans. Although proof scores are executalbe by machines, they are also for human beings to read for proving (or verifying) desired properties on a system of interest. The technique of proof scores was brought up by the OBJ/CafeOBJ community, and substantial developments were done after a reliable implementation of CafeOBJ language system was available. This paper give an overview of evolution of proof scores which have been done under the efforts of verifying vearious kinds of formal specifications in CafeOBJ .

1 Introduction

The need of constructing specifications and verifying them in upstream of software development is still increasing. It is because quite a few critical bugs are caused at the level of domains, requirements, and/or designs specifications. Constructions of specifications are also important for the cases where no program codes are generated but specifications are constructed, analyzed, and verified only for justifying models of real problems.

The goal of verification in software engineering is to increase confidence in the correctness of computer-based systems. For software verification to be genuinely useful, careful account must be taken of the context of actual use, including the goals of the system, the underlying hardware, and the capabilities and culture of users. Absolute certainty is not achievable in real applications, due to the uncertainties inherent in implementation and use, e.g., breakdowns in physical infrastructure (such as cables and computers), errors by human operators, and flaws in connected systems (e.g., operating systems and databases). In this context there is no "silver bullet", but formal methods are still expected to improve the practice of constructions/analyses/verifications of domain/requirement/design specifications.

The term "verify" is used in a sentence like "verify a program against a specification". The term "validate" is used in a sentence like "validate specifications against the reality". This suggests that there is a tradition of making distinction

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 1–20, 2010.
© Springer-Verlag Berlin Heidelberg 2010

between "verification" and "validation": verification means showing by formal proof that some property hold against already formal descriptions, and validation means showing by empirical/experimental means that some property hold against the real world. Following this tradition strictly, verification of domain or requirement specifications does not make sense, for the main characteristic of domain and requirement specifications is in describing systems in the real world and they are difficult to be verified formally.

However, it has turned out that traditional technology of validating informal specification against the reality by means of empirical/experimental way does not meet the current requirement of realizing more safe and secure systems, and a main purpose of verification with proof scores is to overcome this situation and cope with "validation of domain/requirement/design specifications against reality". For making verification to be meaningful also as validation, it should be interactive, for validation against the reality is better to be done by human stakeholders through interactions. Proof scores in CafeOBJ target on more effective and usable interactive verification/validation of high-level (i.e., domain, requirement, or design) specifications.

It is important to distinguish "system specifications" and "property specifications". System specifications are specifications of systems which supposed to be modeled, verified, and/or developed. Property specifications are specifications of properties which is supposed to be satisfied by systems. In [46], the merit of adopting "computational (or executable) specifications" as system specification is advocated. Specifications in CafeOBJ are executable, and both of system and property specifications can be written in equational specifications and executable by interpreting equations as rewriting rules. This makes foundation for verification with proof scores.

The rest of the paper is organized as follows. Section 2 describes development of proof scores in CafeOBJ . Section 3 gives an overview of main features of CafeOBJ language. Section 4 give a complete list of sound and complete proof rules which is an important fundation of proof score constructions. Section 5 describes one of the most recent techniques of combining inference and search in proof scores. This paper is a progress report on our activities on verifications with proof scores, and some parts are revised and/or extended version of contents of our preious papers [14, 15, 19] of siminlar nature.

2 Proof Scores in CafeOBJ

Fully automatic theorem provers often fail to convey an understanding of their proofs, and they are generally unhelpful when they fail because of user errors in specifications or goals, or due to the lack of a necessary lemma or case splitting (both of which are very common in practice). It follows that one should seek to make optimal use of the respective abilities of humans and computers, so that computers do the tedious formal calculations, and humans do the high level planning; the tradeoff between detail and comprehension should also be carefully balanced.

Proof scores are intended to meet these goals [14, 15, 19, 21, 27, 58]. In proof score approach, an executable algebraic specification language (i.e. CafeOBJ in our case) is used to specify systems and system properties, and a processor, i.e. rewrite engine or reducer, of the language is used as a proof engine to prove that the systems satisfy the system properties. Proof plans are coded into proof scores, and are also written in the algebraic specification language. Proof scores are executed by the rewrite engine, and if everything is as expected, an intended proof has been successfully done. Logical soundness of this procedure is guaranteed by the fact that rewritings/reductions done by the rewrite engine is honest to equational axioms of original specifications. Although proof scores can be used to verify code in an imperative language (e.g., as in [33]), it generally makes more sense to verify domain, requirement, or design rather than code.

Proof scores can be regarded as functional programs to prove that a specification satisfy some interesting property. By way of interactions during the development of the proof scores, the quality of the original specification improves profoundly,[55, 56]. The concept of proof supported by proof scores is similar to that of LP [37]. Proof scripts written in a tactic language provided by proof assistants such as Coq [1] and Isabel/HOL [50] have similar nature as proof scores. However, one of unique features of proof scores is that proof scores should constitute a complete document of proof and are supposed to be read by human beings. Moreover, a document (i.e file) of proof scores for proof of a property is intended to be checked its correctness as independently as possible.

Proof score techniques differ from model checking [6] in their emphasis on re-usable domains, requirments, or designs, which may be instantiated in code in many different ways, as well as in their ability to deal with systems that have an infinite number of states, and their natural affinity for abstraction. Many attempts have been done to use model checking techniques to prove designs or algorithms which are expressed as, for example, finite state transition systems. But they are usually expressed only using low-level data types and very close to program code. Besides, model checking only gives "yes" or "no with counter example", and does not support interactive analyses or understandings of domains, requirements, or designs.

2.1 Princiles of Proof Score Approach

The following are major principles which underlie proof score approach.

Human Computer Interaction. Since fully automatic theorem proving is often infeasible for system verification, it is desirable to integrate the human user in the best possible way: in particular, the proof plans or "scores" should be as readable as possible, and helpful feedback should be provided by machines to humans, in order to maximize their ability to contribute.

Flexible but Clear Structure. It is often desirable to arrange the parts of a verification so as to facilitate comprehension. For example, it is often helpful

to state and use a result before it is proved, or even (during proof the planning process) before it is known to be true. Long sequences of proof parts can be difficult to understand, especially when there is no obvious immediate connection between two adjacent parts. But such discontinuities are rather common in published proofs, e.g., when a series of lemmas is proved before they are used in proving a main result. This implies that both subgoals and goal/subgoal relations should be very clear. These flexible and clear structure is coded into several human readable documents of proof scores in CafeOBJ language making uses of its flexible semantics and powerful module facility.

Flexible Logic. It is often possible to simulate one logic within another, by imposing a suitable discipline on how its rules are used; in fact, this is precisely what proof scores accomplish. The choice of underlying basic logics for such a purpose is important in at least three dimensions: its efficiency, its simplicity and ease of use, and its ability to support other logics. We believe that equational logic and its variants are the most suitable for this purpose: Equational logics are relatively simple, have many decidable properties with associated efficient algorithms, are relatively easy to read and write, and can support reasoning in most other logics, e.g., by supplying appropriate definitions to an engine that can apply them by as rewrite rules. By contrast, higher order logic and type theory are much more complex, harder to read and write, harder to mechanize, and harder to reason with, for both humans and machines.

Behavioral Logic. Distributed systems consist of abstract machines in which states are hidden, but can be "observed" by certain "attribute" functions with values in basic data types (such as integer or boolean). Behavioral (also called observational) logic is a natural approach to verifying such systems. Three major approaches in this area are: coalgebra (e.g., see the overview [39]); the "observational logic" of Bidoit and Henniker [2, 38]; and hidden algebra [8, 28], on which our own work is based. We have found a special (but common) case of behavioral modeling scheme in which the effects of methods (or actions) need not be considered for behavioral equivalence, i.e. for which s, s' are behaviorally equivalent if $a(s) = a(s')$ for all applicable attributes a. This is a scheme for concurrent computation similar to **unity**[5], and is named as OTS (Obsevational Transition Systems) [51]. OTS has been extended to TOTS (Timed OTS) [52], which provides a logical basis for verifying real time concurrent systems with proof scores. OTS and TOTS provides powerful modeling scheme for verifications with proof scores.

2.2 Development of Proof Scores

Several well polished small proof scores for data types appeared in OBJ already in 80's, e.g., see [29]. For example, they use reductions to prove induction steps and bases, based on the structure of initial term algebras. Typical examples are proofs of associativity and commutativity of addition for Peano natural numbers, and the identity $n \times (n + 1) = 2 \times (1 + 2 + \cdots + n)$ for any natural number n.

From around 1997, the CafeOBJ group at JAIST [4] started to extend the proof score method (1) to apply to distributed and real-time systems, such as classical distributed (and/or real-time) algorithms, component-based software, railway signal systems, secure protocols, etc., (2) to make the method applicable to practical size problems, and (3) to automate the method. As a result, the proof score method using reduction (rewriting) has become a promising way to do serious proofs.

Many proof scores have been written in CafeOBJ [7, 14] for verifying properties of distributed systems, especially distributed algorithms, component-based software, security protocols, e-commerce protocols, and business workflows [10, 40, 41, 51, 56, 57, 59, 60]. Several auxiliary tools have been also been built to support this progress, including PigNose [47, 48], Gateaux, Crème [49], and Kumo [32].

The following give important developments in verifications with proof scores in CafeOBJ.

From Static to Dynamic Systems. Early proof scores in CafeOBJ included (1) equivalences of functions over natural numbers, (2) equivalences of functions over lists, (3) correctness of simple compilers from expressions to machine code for stack machines, etc. These small proof scores realized an almost ideal combination of high level planning and mechanical reduction. However, even for this class of problems, some non-trivial lemma discovery and/or case spliting is required.

Dynamic systems (i.e. systems with changing states) are common in network/computer based systems, but there is no established methodology in algebraic specification for coping with this class of problems. The CafeOBJ language is designed for writing formal specifications of dynamic systems based on hidden algebra [7, 14, 26]. Many specifications and proof scores for dynamic systems have been done based on hidden algebra semantics, and OTS has been selected as a most promising model. OTS corresponds to a restricted class of hidden algebras, such that it is possible to write specifications for OTS in a fixed standard style that facilitates the development of specifications, and also helps in writing proof scores, since case splittings can be suggested by the specifications. The following publications show stages in the evolution of proof scores for dynamic systems:

- Specifying and verifying mutual exclusion algorithms by incorporating the **unity** model [51].
- Introduction of a primitive version of OTS [53].
- Introduction of real-time features into OTS/CafeOBJ , and accompanying development of proof scores methodology [23, 52].
- A proper introduction of OTS/CafeOBJ and the related proof score writing method [54, 58].
- Examples of verifications with proof scores in OTS/CafeOBJ [55, 56, 57, 59] (among others).

From Explaining to Doing Proofs. A major factor distinguishing the stages of evolution of proof scores is the extent of automation. This is a most important direction of evolution for proof scores, although full automation is not a goal. Automation by reduction is suitable for a mechanical calculation with a focused role and a clear meaning in the context of a larger reasoning process. Early proof scores assisted verification by doing reductions to prove necessary logical statements for a specification. It is intended to gradually codify as many kinds of logical statements as possible into reductions in CafeOBJ . As one extreme case, a fully automatic verification algorithm for a subset of OTS has been developed. This algorithm is developed at syntactic level of logical formula, and not necessary be a good help for interactive verification of high level specifications. Recent important developments are formalization of sound and complete proof rules and collaboration of interence and search in proof scores. The following publications show the stages of automation of proof scores:

– Mainly used for writing formal specifications, but also for proof scores [51].
– Examples with sufficiently complete proof scores [56, 57, 59].
– An attempt for automating proof scores by PigNose (a resolution based automatic theorem prover) [47, 48].
– A fully automatic (algorithmic) method of verification for a subset of OTS [49][1].
– Identification of several techniques to construct effective proof scores for rewriting proof engine [61].
– Formalization of sound and complete proof rules for reachable models [25] (this topic is explained in Section 4).
– Collaboration of interence and search in proof scores [11, 63] (this topic will be explained in Section 5).

3 An Overview of CafeOBJ Language

This section gives an overview of CafeOBJ algebraic specification language which has been culture medium for proof scores.

Algebraic specification technique is one of the most promising system modeling and specification techniques for a wide area of applications. Algebraic specification technique was introduced around 1975 as a method for modeling and specifying so called **abstract data types**. The substantial research efforts were invested to the wide range of area from the basic algebraic semantics theory to many practical application areas. The achievements around OBJ/CafeOBJ can be found at [13, 16, 18, 22, 24].

CafeOBJ is a modern successor of OBJ language[12, 17, 29] and incorporating several most recent algebraic specification paradigms. Its definition is given in [7], and its implementation is reported in [64].

[1] This work used the Maude rewriting engine [42] because it provides faster associative and commutative rewriting.

The following are the major features of the CafeOBJ language system.

Equational Specification and Programming. This is inherited from OBJ [12, 17, 29] and constitutes the basis of the language, the other features being somehow built on top of it. As with OBJ , CafeOBJ is **executable** (by term rewriting), which gives an elegant declarative way of functional programming, often referred as **algebraic programming.**[2] As with OBJ , CafeOBJ also permits equational specification modulo several equational theories such as associativity, commutativity, identity, idempotence, and combinations between all these. This feature is reflected at the execution level by term rewriting **modulo** such equational theories.

Behavioral Specification. Behavioral specification [8, 28] provides a novel generalisation of ordinary algebraic specification. Behavioral specification characterises how objects (and systems) **behave**, not how they are implemented. This new form of abstraction can be very powerful in the specification and verification of software systems since it naturally embeds other useful paradigms such as concurrency, object-orientation, constraints, non determinism, etc. (see [8, 28] for details). Behavioral abstraction is achieved by using specification with hidden sorts and a behavioral concept of satisfaction based on the idea of indistinguishability of states that are observationally the same, which also generalises process algebra and transition systems.

In CafeOBJ a special kind of behavioral specification OTS is identified to have a practical importance in developing proof scores. A fundamental methodological insight behind OTS is it is better to prepare sufficient observations to avoid to encounter true behavioral equivalence.

Rewriting Logic Specification. Rewriting logic specification in CafeOBJ is based on a simplified version of Meseguer's **rewriting logic (RWL)** [44] specification framework for concurrent systems which gives a non-trivial extension of traditional algebraic specification towards concurrency. RWL incorporates many different models of concurrency in a natural, simple, and elegant way, thus giving CafeOBJ a wide range of applications. Unlike Maude [43, 44], the current CafeOBJ design does not fully support **labelled** RWL which permits full reasoning about multiple transitions between states (or system configurations), but provides proof support for reasoning about the **existence** of transitions between states (or configurations) of concurrent systems via a built-in predicate (denoted ==>) with dynamic definition encoding both the proof theory of RWL and the user defined transitions (rules) into equational logic.

From a methodological perspective, CafeOBJ develops the use of RWL transitions for specifying and verifying the properties of **declarative encoding of algorithms** (see [7]) as well as for specifying and verifying transition systems. Transition system plays an important role in using search in proof scores (see Section 5).

[2] Although this paradigm may be used as programming, this aspect is still secondary to its specification side.

Module System. The principles of the CafeOBJ module system are inherited from OBJ which builds on ideas first realized in the language Clear [3], most notably institutions [30]. CafeOBJ module system features

- several kinds of imports,
- sharing for multiple imports,
- parameterised programming allowing
  - multiple parameters,
  - views for parameter instantiation,
  - integration of CafeOBJ specifications with executable code in a lower level language
- module expressions.

However, the theory supporting the CafeOBJ module system represents an updating of the original Clear/OBJ concepts to the more sophisticated situation of multi-paradigm systems involving theory morphisms across institution embeddings [9], and the concrete design of the language revise the OBJ view on importation modes and parameters [7].

Type System and Partiality. CafeOBJ has a type system that allows subtypes based on **order sorted algebra** (abbreviated **OSA**) [31, 36]. This provides a mathematically rigorous form of runtime type checking and error handling, giving CafeOBJ a syntactic flexibility comparable to that of untyped languages, while preserving all the advantages of strong typing.

We decided to keep the concrete order sortedness formalism open at least at the level of the language definition. Instead we formulate some basic simple conditions which any concrete CafeOBJ order sorted formalism should obey. These conditions come close to Meseguer's $\mathrm{OSA}^{\mathrm{R}}$ [45] which is a revised version of other versions of order sortedness existing in the literature, most notably Goguen's OSA [31].

CafeOBJ does not directly do partial operations but rather handles them by using error sorts and a sort membership predicate in the style of **membership equational logic** (abbreviated **MEL**) [45]. The semantics of specifications with partial operations is given by MEL.

Logical semantics. CafeOBJ is a declarative language with firm mathematical and logical foundations in the same way as other OBJ -family languages (OBJ, Eqlog [34], **FOOPS**[35], Maude [43, 44]) are. The reference paper for the CafeOBJ mathematical foundations is [9], while [7] gives a somehow less mathematical easy-to-read (including many examples) presentation of the semantics of CafeOBJ .

The mathematical semantics of CafeOBJ is based on algebraic specification concepts and results, and is strongly based on category theory and the theory of institutions [9, 30]. The following are the principles governing the logical and mathematical foundations of CafeOBJ :

P1: There is an underlying logic[3] in which all basic constructs and features of the language can be rigorously explained.
P2: Provide an integrated, cohesive, and unitary approach to the semantics of specification in-the-small and in-the-large.
P3: Develop all ingredients (concepts, results, etc.) at the highest appropriate level of abstraction.

4 Sound and Complete Proof Rules for Reachalbe Models

A formal specification denotes models. That is the meaning of a specification is defined to be the set of models each element of which satisfy the specification. A good formal specification gives clear and transparent view of the entity the specification describes through the models which denotes. Formal proof based on a formal specification just verifies that any model which satisfy the specification satisfy some property of interest.

Verification or proof should be done using sufficiently powerful proof rules. Proof rules are syntactic rules defined in a formal (i.e mathimatical) language for describing specifications and properties.

Let Γ denotes a specifications and ρ denotes a property, then $\Gamma \models \rho$ denotes that any model which satisfy Γ satisfy ρ, and $\Gamma \vdash \rho$ denotes that *rho* can be derived from *Gamma* using proof rules. The binary relation $(_ \vdash _)$ is called *entailment relation*.

Proof rules are called *sound* if the following holds,

$$\Gamma \vdash \rho \text{ implies } \Gamma \models \rho$$

and are called *complete* if the following holds.

$$\Gamma \models \rho \text{ implies } \Gamma \vdash \rho$$

We have develop sound and complete proof rules for specifications which only denotes reachable models [25]. A reachable model is the model which only consists of elements composed of *constructor* operations. Restricting models to reachable models are quite reasonable for many practically cases, and it is a good news that it is possible to have complete (i.e. the most powerful) proof rules for this situation.

The following list gives sound and complete proof rules. Notice that the last two rules are of infinite nature, and these are secret of why this list of rules is complete. Notice also that the entailment relation $(_ \vdash _)$ is defined between two sets of equations. However, depending on context, a single equation can be considered to be a singleton set of the equation, a set of equations can be considered to be an equation:

[3] Here "logic" should be understood in the modern relativistic sense of "institution" which provides a mathematical definition for a logic (see [30]) rather than in the more classical sense.

$$\langle \text{conjunction of all the equations in the set} \rangle = \text{true},$$

and any Boolean expression b can be considered to be an equation ($b = \text{true}$). Intereted readers are advised to look into [25] or the slides for lecture 03b of [11] for details.

Entailment Relation. The following four rules are basics of entailment relation.

(Monotonicity)

$$\frac{}{E_1 \vdash E_2} \text{ whenever } E_2 \subseteq E_1$$

(Transitivity)

$$\frac{E_1 \vdash E_2, E_2 \vdash E_3}{E_1 \vdash E_3}$$

(Unions)

$$\frac{E_1 \vdash E_2, E_1 \vdash E_3}{E_1 \vdash E_2 \cup E_3}$$

(Translation)

$$\frac{E \vdash_{\Sigma} E'}{\varphi(E) \vdash_{\Sigma'} \varphi(E')} \text{ for all signature morphisms } \varphi : \Sigma \to \Sigma'$$

Equality. The following six rules characterize the equality relation.

(Reflexivity)

$$\frac{}{\emptyset \vdash t = t}$$

(Symmetry)

$$\frac{}{t = t' \vdash t' = t}$$

(Transitivity)

$$\frac{}{\{t = t', t' = t''\} \vdash t = t''}$$

(Congruence)

$$\frac{}{\{t_i = t'_i | i = \overline{1,n}\} \vdash \sigma(t_1, ..., t_n) = \sigma(t'_1, ..., t'_n)}$$

(P-Congruence)

$$\frac{}{\{t_i = t'_i | i = \overline{1,n}\} \cup \{\pi(t_1, ..., t_n)\} \vdash \pi(t'_1, ..., t'_n)}$$

Meta and Object Implications. The following gives equivalence of mata and object implications.

(Implications)

$$\frac{\Gamma \vdash \bigwedge H \Rightarrow C}{\Gamma \cup H \vdash C} \text{ and } \frac{\Gamma \cup H \vdash C}{\Gamma \vdash \bigwedge H \Rightarrow C}$$

Substitution and Generalization. Rules for substitution and generalization. Generalization is also called **Theorem of Constant**

(Substitutivity)

$$\frac{}{(\forall x)\rho \vdash (\forall Y)\rho(x \leftarrow t)}$$

(Generalization)

$$\frac{\Gamma \vdash_{\Sigma} (\forall Z)\rho}{\Gamma \vdash_{\Sigma(Z)} \rho} \quad \textit{and} \quad \frac{\Gamma \vdash_{\Sigma(Z)} \rho}{\Gamma \vdash_{\Sigma} (\forall Z)\rho}$$

Infinite Rules. The following two rules are of infinite nature, and justify the two of the important verification techniques of *induction* and *case analysis* (or *case splitting*).

(C-Abstraction)

$$\frac{\{\Gamma \vdash_{\Sigma} (\forall Y)\rho(x \leftarrow t) \mid Y \text{ are loose vars, } t \text{ is constructor } Y\text{-term}\}}{\Gamma \vdash_{\Sigma} (\forall x)\rho}$$

(Case Analysis)

$$\frac{\{\Gamma \cup \{\sigma(t_1, \ldots, t_n) = t\} \vdash_{\Sigma(Y)} e \mid \ Y \text{ are loose vars, } t \text{ is constructor } Y\text{-term}\}}{\Gamma \vdash_{\Sigma} e}$$

5 Combining Inference and Search with Proof Scores

Verification with proof scores can be quite versatile, and realizes powerful techniques of combining inference and search for proving interesting properties of interest. This section gives an overview of an example of verifying mutual exclusion property with combination of inference and search. You can find the more detailed explanation with complete CafeOBJ codes of this example in the lecture materials of Lecture05a and Lecture09a at the web page of [11]. You can also find another kind of combination of inference and search with proof scores in [63].

Mutual Exclusion Protocols. Mutual excusion protocols are described as follows:

> *Assume that many agents (or processes) are competing for a common equipment, but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (mechanism or algorithm) which can achieve the mutual exclusion is called mutual exclusion proto that many agents (or processes) are competing for a common equipment, but at any moment of time only one agent can use the equipment. That is, the agents are mutually excluded in using the equipment. A protocol (mechanism or algorithm) which can achieve the mutual exclusion is called* mutual exclusion protocol.

A typical mutual exclusion protocol, called QLOCK, is realized by using a global queue as follows.

- Each of unbounded number of agents who participates the protocol behaves as follows:
  - If the agent wants to use the common equipment and its name is not in the queue yet, put its name into the queue.
  - If the agent wants to use the common equipment and its name is already in the queue, check if its name is on the top of the queue. If its name is on the top of the queue, start to use the common equipment. If its name is not on the top of the queue, wait until its name is on the top of the queue.
  - If the agent finishes to use the common equipment, remove its name from the top of the queue.
- The protocol should start from the state where the queue is empty.
- It is assumed that each agent can use the common equipment any number of times.

QLOCK in OTS/CafeOBJ. In OTS a system is modeled as a transition system whose state is identified as a collection of typed values given by *observation* operations. A state transition of the system is modeled as an *action* operation which returns next state.

QLOCK is modeled in OTS (Observational Transition System) as Fig. 1, and Fig. 2 gives CafeOBJ signature for QLOCK in OTS .

QLOCK is modeled as OTS with two observations of `pc`, `queue` and three actions of `want`, `try`, `exit`. Initial states are modeled as a constant (i.e. an operation with no arguments) `init`.

`Sys` is sort of system space, and `Pid` is sort of agent names. `pc` returns one of three labels of `rm`, `wt`, `cs` to indicate the state of each agent. `queue` returns a value of the global queue.

Behavior of an OTS is specified by giving equations which define the way in which each action changes the values of observations. For example, the specification of the action of `want` of QLOCK is described in CafeOBJ as follows.

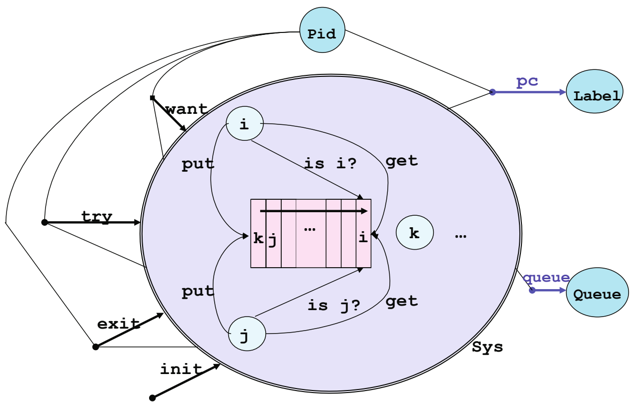


Fig. 1. Global view of QLOCK in Observational Transition System

```
-- state space of the system
[Sys]                                       system sort declaration

-- visible sorts for observation
[Queue Pid Label]                           visible sort declaration

-- observations
op pc : Sys Pid -> Label                    observation declaration
op queue : Sys -> Queue

-- any initial state
op init : -> Sys (constr)
-- actions
op want : Sys Pid -> Sys {constr}
op try  : Sys Pid -> Sys {constr}           action declaration
op exit : Sys Pid -> Sys {constr}
```

Fig. 2. CafeOBJ signature for QLOCK in Observational Transition System

```
-- want
op c-want : Sys Pid -> Bool
eq c-want(S:Sys,I:Pid) = (pc(S,I) = rm).
--
ceq pc(want(S:Sys,I:Pid),J:Pid)
     = (if I = J then wt else pc(S,J) fi)      if c-want(S,I).
ceq queue(want(S:Sys,I:Pid)) = put(I,queue(S)) if c-want(S,I).
ceq want(S:Sys,I:pid)        = S               if not c-want(S,I).
```

`rm` is a label showing an agent is not in the global queue, and `(pc(S,I) = rm)` indicates that the agent `I` is not in the global queue in the state `S`. `c-want` is

a predicate (i.e. function with Boolean value) defining the condition only under which the action `want` changes state (i.e. values of observations). That is, if `c-want` is false then `want` does not change any observations. The first conditional equation (i.e. `ceq`) defines the change of value of observation `pc`, and the second conditional equation defines the change of value of observation `queue`. The third conditional equation defines that state does not change if `c-want` is false.

The specifications for other two actions `try`, `exit` are given in similar ways. Specifications of these three actions are the core part of QLOCK specification, but it is important to notice that they are based on precise specification of the global queue, e.g. the specification of operation `put`.

It is also important to notice that by the style of defining behaviors actions of OTS using conditional equations, any.

Let a predicate `mex` be defined as follows, where `(pc(S,I) = cs)` means that an agent `I` is using the common equipment at the state `S`. The proposition `mex(s,i,j)` means that if two agents of `i` and `j` are using the common equipment then `i` and `j` should be identical.

```
eq mex(S:Sys,I:Pid,J:Pid)
   = (((pc(S,I) = cs) and (pc(S,J) = cs)) implies I = J) .
```

Let `QLOCK` be the module name of CafeOBJ specification which specifies QLOCK protocol and also includes the above predicate `mex`. The mutual exclusion property to be verified on `QLOCK` is expressed as follows. This formula means that for any model which satisfies the specification `QLOCK` , the proposition $\mathtt{mex}(s, i, j)$ should be true for any s:`Sys`, i:`Pid`, j:`Pid`.

$$\mathtt{QLOCK} \models \forall S : \mathtt{Sys}\, \forall I : \mathtt{Pid}\, \forall J : \mathtt{Pid}\, .\, \mathtt{mex}(S, I, J)$$

Verifying QLOCK with Small Number of Agents by Search. The sort `Sys` of `QLOCK` is defined to be the set of all states which can be reached from the initial state `init` via three actions of `want`, `try`, and `exit`. That is, `Sys` can be defined as follows.

$$\begin{aligned}\mathtt{Sys} \stackrel{\text{def}}{=} & \{\mathtt{init}\} \cup \\ & \{\mathtt{want}(s, i) \mid s : \mathtt{Sys}, i : \mathtt{Pid}\} \cup \\ & \{\mathtt{try}(s, i) \mid s : \mathtt{Sys}, i : \mathtt{Pid}\} \cup \\ & \{\mathtt{exit}(s, i) \mid s : \mathtt{Sys}, i : \mathtt{Pid}\}\end{aligned}$$

If the number of agents who participate QLOCK protocol is limited to a small number, it is feasible to search all possible states and check whether the mutual exclusion property $\mathtt{mex}(s, i, j)$ holds for any state s:`Sys`.

In CafeOBJ , transitions can be defined and built-in search predicate provides the facility to check whether a property holds at all reachable states via the transitions.

A transition system corresponding to QLOCK is described as follows in CafeOBJ, where S is a CafeOBJ variable standing for any state and p is a CafeOBJ constant standing for arbitrary agent.

```
ctrans < S > => < want(S,p) > if c-want(S,p) .
ctrans < S > => < try(S,p) >  if c-try(S,p) .
ctrans < S > => < exit(S,p) > if c-exit(S,p) .
```

Let QLOCKpTrans denote a CafeOBJ module with above three conditional transitions which defines a transition system corresponding to QLOCK. The transition system with two agents can be specified as follows using powerful module expression of CafeOBJ .

```
-- transition system with 2 agents i j
mod* QLOCKijTrans {
 inc((QLOCKpTrans * {op p -> i}) +
     (QLOCKpTrans * {op p -> j})) }
```

It is easy by the above definition of Sys to see that the Sys (the reachable state space) of QLOCKijTrans can be spanned by six conditonal transitions of QLOCKijTrans from the initial state init.

CafeOBJ provides the following built-in predicate, where $t1$ and *t2* are expressions to represent initial and final states, *it* m is a natural number which limits number of final states to be found, n is a natural number which limits number of transitions, $pred1(t2)$ is a predicate on $t2$ specifying the final states to be found, and $pred2(S1 : State, S2 : State)$ is a binary predicate on the sort of $t1$ and $t2$ specifying identity relation which is used to stop searching when encounters identical one. m and n can be * for specifying "no limit".

$$t1 = (m, n) => *\, t2 \;\texttt{suchThat}\; pred1(t2) \;\; \texttt{withStateEq}\; pred2(S1 : State, S2 : State)$$

Using the above built-in predicate, the mutual exclusion property of the two agents QLOCL protocol is verified by checking that the following reduction returns false. Where, Config is the sort of state space of QLOCKijTrans, and (C1:Config =ob= C2:Config) is binary predicate over Config to check whether two states return the same observations.

```
red < init > =(*,*)=>* < S:Sys >
          suchThat (not mex(S,i,j))
          withStateEq (C1:Config =ob= C2:Config) .
```

Verifying Simulation of Any Number of Agents by Two Agents. If the behaviors of QLOCK (with any number of agents) which relate to an interested property can be simulated by QLOCK with two agents, the property can be verified by only inspecting the behavior of the two agents QLOCK. This observation can be formalized as follows.

Let i and j denote two distinct agent names, and $\mathtt{Sys}(i,j)$ denote the state space spanned only by actions caused by these two agents, that is $\mathtt{Sys}(i,j) \subseteq \mathtt{Sys}$. Simulation[4] of $\mathtt{Sys}$ by $\mathtt{Sys}(i,j)$ is defined as follows.

Proposition-A. For any reachable state $s : \mathtt{Sys}$ there exit a reachable state $t : \mathtt{Sys}(i,j)$ which satisfy the following property.

$$\begin{aligned}\mathtt{sim}(s,i,j,t) \stackrel{\text{def}}{=} \ & (\mathtt{pc}(s,i) = \mathtt{pc}(t,i)) \ \mathit{and} \\ & (\mathtt{pc}(s,j) = \mathtt{pc}(t,j)) \ \mathit{and} \\ & (\mathtt{pr}_{ij}(\mathtt{queue}(s)) = \mathtt{queue}(t))\end{aligned}$$

Where $\mathtt{pr}_{ij}(Q : \mathtt{Queue})$ is defined as follows.

$$\begin{aligned}\mathtt{pr}_{ij}(\mathtt{nil}) &= \mathtt{nil} \\ \mathtt{pr}_{ij}(Q : \mathtt{Queue}, I : \mathtt{Pid}) &= \begin{cases} (\mathtt{pr}_{ij}(Q), I) & \mathtt{if}\ ((I = i)\ \mathtt{or}\ (I = j)) \\ \mathtt{pr}_{ij}(Q) & \mathtt{if}\ \mathtt{not}((I = i)\ \mathtt{or}\ (I = j)) \end{cases}\end{aligned}$$

Where $(_,_)$ is the cons(truction) operation of $\mathtt{Queue}$. □

Proof scores can be developed to verify **Proposition-A**. It is easy to see the following predicate hold, and proof scores can also be developed for this.

$$(\mathtt{sim}(s,i,j,t)\ \mathtt{and}\ \mathtt{mex}(t,i,j))\ \mathtt{implies}\ \mathtt{mex}(s,i,j)$$

We have already verified that $\mathtt{mex}(t,i,j)$ hold for any state t of two agents QLOCK, and $\mathtt{sim}(s,i,j,t)$ means that such t exits for any state s of any number agents QLOCK, this implies that $\mathtt{mex}(s,i,j)$ hold for any reachable state s of QLOCK.

6 Conclusions

This paper gives an overview of evolution of verifications with proof scores in CafeOBJ . Specifications in CafeOBJ can be expressed in relatively high level of abstraction thanks to facilities of user defined data types, OTS , and powerful module systems, etc. Proof scores enjoy the same merit and can make it possible to verify the specifications or domains, requirements, designs in a more higher level of abstraction than other interactive theorem provers or model checkers. This suggests that proof score approach has a potential for providing a practical way of verifying specifications.

There are several challenges to be attacked in the future. The following are most important current issues of the proof score approach with CafeOBJ :

- Development of correctness checker for proof scores which can check the correctness as independently as possible. That is, using minimal semantical informal about the original specification.

[4] This definition of simulation is deferent from that of [62], but has a similar intention.

- Farther development of the Kumo/Tatami project (done at UCSD as a sub-project of CAFE project [21]) to realize a web (or hypertext) based proof score development environment.
- Serious development of practical domain and/or requirement specifications in the application area like e-government, e-commerce, open standards for automotive software, etc. The development should aim at a reasonable balance of informal and the formal specifications, and verify as much as meaningful and important properties of the models/problems the specifications are describing.

Acknowledgments. The achievements reported in this paper are done by many people almost all names of whom can be found as the authors of publications listed at the following References.

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004)
2. Bidoit, M., Hennicker, R.: Behavioural theories and the proof of behavioural properties. Theoretical Computer Science 165(1), 3–55 (1996)
3. Burstall, R.M., Goguen, J.A.: Putting theories together to make specifications. In: IJCAI, pp. 1045–1058 (1977)
4. CafeOBJ: Web page (2010), http://www.ldl.jaist.ac.jp/cafeobj/
5. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
6. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (2000)
7. 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)
8. Diaconescu, R., Futatsugi, K.: Behavioural coherence in object-oriented algebraic specification. Journal of Universal Computer Science 6(1), 74–96 (2000)
9. Diaconescu, R., Futatsugi, K.: Logical foundations of CafeOBJ. Theor. Comput. Sci. 285(2), 289–318 (2002)
10. Diaconescu, R., Futatsugi, K., Iida, S.: Component-based algebraic specification and verification in CafeOBJ. In: Wing et al. [65], pp. 1644–1663
11. FSSV 2010: Web page (March 2010), http://www.ldl.jaist.ac.jp/jaist-fssv2010/lectureMaterials/
12. Futatsugi, K.: An overview of OBJ2. In: Fuchi, K., Nivat, M. (eds.) Programming of Future Generation Computers, pp. 139–160. North-Holland, Amsterdam (1986); Proc. of Franco-Japanese Symp. on Programming of Future Generation Computers, Tokyo (October 1986)
13. Futatsugi, K.: Trends in formal specification methods based on algebraic specification techniques – from abstract data types to software processes: A personal perspective. In: Proceedings of the International Conference of Information Technology to Commemorating the 30th Anniversary of the Information Processing Society of Japan (InfoJapan 1990), pp. 59–66. Information Processing Society of Japan (1990) (invited talk)

14. Futatsugi, K.: Formal methods in CafeOBJ. In: Hu, Z., Rodríguez-Artalejo, M. (eds.) FLOPS 2002. LNCS, vol. 2441, pp. 1–20. Springer, Heidelberg (2002)
15. Futatsugi, K.: Verifying specifications with proof scores in CafeOBJ. In: Proc. of 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), pp. 3–10. IEEE Computer Society, Los Alamitos (2006)
16. Futatsugi, K., Goguen, J., Meseguer, J. (eds.): OBJ/CafeOBJ/Maude at Formal Methods 1999. The Theta Foundation, Bucharest (1999) ISBN 973-99097-1-X
17. Futatsugi, K., Goguen, J.A., Jouannaud, J.P., Meseguer, J.: Principles of OBJ2. In: Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL 1985), New Orleans, Louisiana, pp. 52–66. ACM, New York (1985)
18. Futatsugi, K., Goguen, J.A., Meseguer, J., Okada, K.: Parameterized programming in OBJ2. In: ICSE, pp. 51–60 (1987)
19. Futatsugi, K., Goguen, J.A., Ogata, K.: Verifying design with proof scores. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, Springer, Heidelberg (2008)
20. Futatsugi, K., Jouannaud, J.P., Meseguer, J. (eds.): Algebra, Meaning, and Computation. LNCS, vol. 4060. Springer, Heidelberg (2006)
21. Futatsugi, K., Nakagawa, A.: An overview of CAFE specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In: Proc. of 1st International Conference on Formal Engineering Methods (ICFEM 1997), Hiroshima, Japan, November 12-14, pp. 170–182. IEEE, Los Alamitos (1997)
22. Futatsugi, K., Nakagawa, A., Tamai, T. (eds.): CAFE: An Industrial-Strength Algebraic Formal Method. Elsevier Science B.V., Amsterdam (2000) ISBN 0-444-50556-3
23. Futatsugi, K., Ogata, K.: Rewriting can verify distributed real-time systems. In: Proc. of International Symposium on Rewriting, Proof, and Computation (PRC 2001), pp. 60–79. Tohoku Univ. (2001)
24. Futatsugi, K., Okada, K.: Specification writing as construction of hierarchically structured clusters of operators. In: IFIP Congress, pp. 287–292 (1980)
25. Gâinâ, D., Futatsugi, K., Ogata, K.: Constructor-based institutions. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 398–412. Springer, Heidelberg (2009)
26. Goguen, J.: Hidden algebraic engineering. In: Nehaniv, C., Ito, M. (eds.) Algebraic Engineering. pp. 17–36. World Scientific, Singapore (1998), papers from a conference at the University of Aizu, Japan, 24–26 March 1997; also UCSD Technical Report CS97–569 (December 1997)
27. Goguen, J.: Theorem Proving and Algebra. [Unpublished Book] (now being planned to be up on the web for the free use)
28. Goguen, J., Malcolm, G.: A hidden agenda. Theoretical Computer Science 245(1), 55–101 (2000)
29. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.P.: Introducing OBJ. In: Goguen, J., Malcolm, G. (eds.) Software Engineering with OBJ: Algebraic Specification in Action, pp. 3–167. Kluwer, Dordrecht (2000)
30. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. J. ACM 39(1), 95–146 (1992)
31. Goguen, J.A., Diaconescu, R.: An oxford survey of order sorted algebra. Mathematical Structures in Computer Science 4(3), 363–392 (1994)

32. Goguen, J.A., Lin, K., Mori, A., Rosu, G., Sato, A.: Distributed cooperative formal methods tools. In: Proc. of 1997 International Conference on Automated Software Engineering (ASE 1997), Lake Tahoe, CA, November 02-05, pp. 55–62. IEEE, Los Alamitos (1997)
33. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. MIT Press, Cambridge (1996)
34. Goguen, J.A., Meseguer, J.: Eqlog: Equality, types, and generic modules for logic programming. In: DeGroot, D., Lindstrom, G. (eds.) Logic Programming: Functions, Relations, and Equations, pp. 295–363. Prentice-Hall, Englewood Cliffs (1986)
35. Goguen, J.A., Meseguer, J.: Unifying functional, object-oriented and relational programming with logical semantics. In: Shriver, B., Wegner, P. (eds.) Research Directions in Object-Oriented Programming, pp. 417–478. MIT Press, Cambridge (1987)
36. Goguen, J.A., Meseguer, J.: Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. Theor. Comput. Sci. 105(2), 217–273 (1992)
37. Guttag, J.V., Horning, J.J., Garland, S.J., Jones, K.D., Modet, A., Wing, J.M.: Larch: Languages and Tools for Formal Specification. Springer, Heidelberg (1993)
38. Hennicker, R., Bidoit, M.: Observational logic. In: Haeberer, A.M. (ed.) AMAST 1998. LNCS, vol. 1548, pp. 263–277. Springer, Heidelberg (1998)
39. Jacobs, B., Rutten, J.: A tutorial on (co)algebras and (co)induction. Bulletin of the European Association for Theoretical Computer Science 62, 222–259 (1997)
40. Kong, W., Ogata, K., Futatsugi, K.: Algebraic approaches to formal analysis of the mondex electronic purse system. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 393–412. Springer, Heidelberg (2007)
41. Kong, W., Ogata, K., Futatsugi, K.: Specification and verification of workflows with Rbac mechanism and Sod constraints. International Journal of Software Engineering and Knowledge Engineering 17(1), 3–32 (2007)
42. Maude: Web page (2010), http://maude.cs.uiuc.edu/
43. Meseguer, J.: A logical theory of concurrent objects. In: OOPSLA/ECOOP, pp. 101–115 (1990)
44. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
45. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
46. Meseguer, J.: From OBJ to Maude and beyond. In: Futatsugi et al. [20], pp. 252–280
47. Mori, A., Futatsugi, K.: Verifying behavioural specifications in CafeOBJ environment. In: Wing et al. [65], pp. 1625–1643
48. 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. 461–470. Springer, Heidelberg (2003)
49. Nakano, M., Ogata, K., Nakamura, M., Futatsugi, K.: Crème: an automatic invariant prover of behavioral specifications. International Journal of Software Engineering and Knowledge Engineering 17(6), 783–804 (2007)
50. Nipkow, T., Paulson, L.C., Wenzel, M.T. (eds.): Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)

51. Ogata, K., Futatsugi, K.: Specification and verification of some classical mutual exclusion algorithms with CafeOBJ. In: Futatsugi et al. [16], pp. 159–177 ISBN 973-99097-1-X
52. Ogata, K., Futatsugi, K.: Modeling and verification of distributed real-time systems based on CafeOBJ. In: Proceedings of the 16th International Conference on Automated Software Engineering (16th ASE), pp. 185–192. IEEE Computer Society Press, Los Alamitos (2001)
53. Ogata, K., Futatsugi, K.: Specifying and verifying a railroad crossing with CafeOBJ. In: Proceedings of the 6th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (6th FMPPTA); Part of Proceedings of the 15th IPDPS, p. 150. IEEE Computer Society Press, Los Alamitos (2001)
54. Ogata, K., Futatsugi, K.: Rewriting-based verification of authentication protocols. In: 4th WRLA. ENTCS, vol. 71, pp. 189–203. Elsevier, Amsterdam (2002)
55. Ogata, K., Futatsugi, K.: Flaw and modification of the iKP electronic payment protocols. Information Processing Letters 86(2), 57–62 (2003)
56. Ogata, K., Futatsugi, K.: Formal analysis of the iKP electronic payment protocols. In: Okada, M., Pierce, B.C., Scedrov, A., Tokuda, H., Yonezawa, A. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 441–460. Springer, Heidelberg (2003)
57. Ogata, K., Futatsugi, K.: Formal verification of the Horn-Preneel micropayment protocol. In: Zuck, L.D., Attie, P.C., Cortesi, A., Mukhopadhyay, S. (eds.) VMCAI 2003. LNCS, vol. 2575, pp. 238–252. Springer, Heidelberg (2002)
58. 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)
59. Ogata, K., Futatsugi, K.: Equational approach to formal verification of SET. In: Proceedings of the 4th International Conference on Quality Software (4th QSIC), pp. 50–59. IEEE Computer Society Press, Los Alamitos (2004)
60. Ogata, K., Futatsugi, K.: Equational approach to formal analysis of TLS. In: 25th ICDCS, pp. 795–804. IEEE CS Press, Los Alamitos (2005)
61. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. In: Futatsugi et al. [20], pp. 596–615
62. Ogata, K., Futatsugi, K.: Simulation-based verification for invariant properties in the ots/cafeobj ethod. Electr. Notes Theor. Comput. Sci. 201, 127–154 (2008)
63. Ogata, K., Futatsugi, K.: A combination of forward & backward reachability analysis method. In: Proc. of 12th International Conference on Formal Engineering Methods (ICFEM 2010), Shanghai, China, November 16-19. Springer, Heidelberg (2010) (to appear)
64. Sawada, T., Kishida, K., Futatsugi, K.: Past, present, and future of SRA implementation of CafeOBJ: Annex. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 7–17. Springer, Heidelberg (2003)
65. Wing, J.M., Woodcock, J., Davies, J. (eds.): FM 1999. LNCS, vol. 1709. Springer, Heidelberg (1999)

Exploiting Partial Success in Applying Automated Formal Methods

Matthew B. Dwyer

Department of Computer Science and Engineering, University of Nebraska,
Lincoln NE 68588-0115, USA
dwyer@cse.unl.edu

Abstract. The past decades have produced a wide-range of formal techniques for developing and assessing the correctness of software systems. Techniques, such as various forms of static analysis, automated verification, and test generation, can provide valuable information about how a system satisfies its specified correctness properties. In practice, when applied to large modern software systems all existing automated formal methods come up short. They might produce false error reports, exhaust available human or computational resources, or be incapable of reasoning about some set of important properties. Whatever their shortcoming, the goal of *proving* a system correct remains elusive.

Despite this somewhat dire outlook, there have been enormous gains in the effectiveness of a range of automated formal methods. Rather than looking for a *silver bullet* of a formal method, we ought to admit that no one method will be effective for all properties on all software systems. We should embrace the wealth of existing techniques by trying to characterize their relative strengths and weaknesses across a range of properties and software domains.

Moreover, we should exploit the conventional wisdom that software systems are *mostly correct* – systems have much more correct behavior than incorrect behavior. Given this we should shift from focusing on proving correctness, to developing automated formal methods that calculate the set of system behaviors that are consistent with system specifications. Clearly if the specification-consistent set of behaviors is the set of all behaviors, then the property is proved, but that will rarely be the case.

It is likely, however, that methods will be able to demonstrate that large sets of behaviors are specification-consistent. This type of *partial evidence* of correctness will be most valuable if evidence from multiple techniques can be combined. Equipped with a rich suite of evidence-producing formal methods, where the weakness of each method is masked by the strength of another, and a means for combining their partial evidence we will be well positioned to target the verification and validation of modern software systems.

Acknowledgments

We thank Elena Sherman and Sebastian Elbaum for discussions related to this work. This work was supported in part by the National Science Foundation through awards CCF-0747009 and CCF-0915526, the National Aeronautics and Space Administration under grant number NNX08AV20A, and the Air Force Office of Scientific Research under award FA9550-09-1-0129.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, p. 21, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Multicore Embedded Systems: The Timing Problem and Possible Solutions

Wang Yi

Uppsala University, Sweden

Today's processor chips contain often multiple CPUs i.e. processor cores each of which may support several hardware threads working in parallel. They are known as multicore or many-core processors. As a consequence of the broad introduction of multicore into computing, almost all software must exploit parallelism to make the most efficient use of on-chip resources including processor cores, caches and memory bandwidth. For embedded applications, it is predicted that multicores will be increasingly used in future embedded systems for high performance and low energy consumption. The major obstacle is that due to on-chip resource contention, the prediction of system performance, latencies, and resource utilization in multicore systems becomes a much harder task than that for single-core systems. With the current technology we may not predict and provide any guarantee on real-time properties of multicore software, which restricts seriously the use of multicores for embedded applications.

In this talk, I will give an overview on the key challenges for software development on multicore architecture and briefly introduce the CoDeR-MP [1] project at Uppsala to develop high-performance and predictable real-time software on multicore platforms. I will present the multicore timing analysis problem and our solutions proposed in a series of recent work. Technical details may be found in [LNYY10] on combining abstract interpretation and model checking for multicore WCET analysis, [GSYY09a] dealing with shared caches, [GSYY09b] on response time analysis for multicore systems, and [GSYY10] extending Layland and Liu's classical result [LL73] on rate monotonic scheduling for single-core systems to multicore systems.

References

[GSYY09a] Guan, N., Stigge, M., Yi, W., Yu, G.: Cache-aware scheduling and analysis for multicores. In: ACM Conference on Embedded Software (EMSOFT), pp. 245–254 (2009)

[GSYY09b] Guan, N., Stigge, M., Yi, W., Yu, G.: New response time bounds for fixed priority multiprocessor scheduling. In: IEEE Real-Time Systems Symposium (RTSS), pp. 387–397 (2009)

[1] CoDeR-MP: Computationally Demanding Real-Time Applications on Multicore Platforms, www.it.uu.se/research/coder-mp, a joint research program at Uppsala University with ABB and SAAB, supported by the Swedish Foundation for Strategic Research.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 22–23, 2010.
© Springer-Verlag Berlin Heidelberg 2010

[GSYY10] Guan, N., Stigge, M., Yi, W., Yu, G.: Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. In: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pp. 165–174 (2010)

[LL73] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM 20(1), 46–61 (1973)

[LNYY10] Lv, M., Nan, G., Yi, W., Yu, G.: Combining abstract interpretation with model checking for timing analysis of multicore software. In: IEEE Real-Time Systems Symposium, RTSS (to appear 2010)

Applying PVS Background Theories and Proof Strategies in Invariant Based Programming

Johannes Eriksson and Ralph-Johan Back

Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5 A, Turku, FI-20520, Finland
{joheriks,backrj}@abo.fi

Abstract. *Invariant Based Programming (IBP)* is a formal method in which the invariants are developed before the code. IBP leads to programs that are correct by construction, provides a light formalism that is easy to learn, and supports teaching formal methods. However, like other verification methods it generates a large number of lemmas to be proved. The *Socos tool* provides automatic verification of invariant based programs: it sends the proof obligations to an automatic theorem prover and reports only the unproven conditions. The latter may be proved interactively in a proof assistant.

In this paper, we describe the Socos embedding of invariant based programs into the theorem prover PVS. The tool generates verification conditions and applies a strategy to decompose the conditions into fine grained lemmas. Each lemma is then attacked with the SMT solver Yices. Socos supports incremental development and allows reasoning in arbitrary program domains through the use of *background theories*. A background theory is a PVS theory pertaining to a specific programming domain. We give an example of a verification in our system, which demonstrates how background theories improve the degree of proof automation. This work is a step towards scaling up IBP by allowing existing collections of PVS theories to be used.

1 Introduction

Building correct programs is a core problem in computer science. Formal methods for verifying with mathematical rigor that a program satisfies a specification have been proposed to address the issue. *Invariant Based Programming (IBP)* is a new correct-by-construction formal method, in which not only the pre- and postconditions, but also the loop invariants are written before the code [1].

Tool support is key to making formal methods feasible. A program verifier generates verification conditions (VCs) and applies an automatic theorem prover (ATP) to them. Modern ATPs, such as satisfiability modulo theories (SMT) solvers efficiently discharge large numbers of simple conditions. IBP is supported by *Socos* [2], a diagrammatic environment connected to a VC generator and an ATP. Its core is a semantics for translating invariant based programs into VCs, which are sent to an SMT solver for simplification. The programmer needs to consider only the conditions that were not proved automatically.

All conditions cannot be expected to be proved automatically, even for small programs. In theory, the remaining conditions could be proved interactively using a proof assistant. In practice, proving requires considerable expertise, and is hard to integrate

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 24–39, 2010.
© Springer-Verlag Berlin Heidelberg 2010

into the programming workflow as program proofs tend to be tedious and must be rechecked whenever the program is modified. Hence interactive verification is mainly used in safety-critical and other dependable systems. An alternative approach explored here consists of identifying and proving the central theorems on which the correctness of the program hinges, and then using strong automation to apply these as lemmas in the proofs of the concrete VCs. The first activity is part of developing the *background theory*—a mathematical theory pertaining to the program domain at hand. In this way the programmer can focus on the essential conditions, and hand over the details to the ATP. Too much detail can swamp ATPs, so proofs must be carried out at a proper level of abstraction for efficient automation. In this paper we study the application of the PVS theorem prover [13] and the Yices SMT solver [9] in IBP, and how automatic and interactive verification are combined in the Socos environment. By defining the necessary abstractions in a PVS background theory and sending them as axioms to an SMT solver, a high level of proof automation can be attained for programs in the domain at hand.

Invariant diagrams. IBP uses the basic building block of a *situation*, a collection of constraints to describe a set of states. Situations can be *nested* to inherit the constraints of outer situations. A *transition* is a program statement going from one situation to another. An *initial* situation has no incoming transitions; a *final* situation has no outgoing transitions. *Invariant diagrams* are graphs of situations and transitions: the former are drawn as rounded boxes, the latter as arrows connecting to the edges of the boxes. Figure 1 shows an example: a program that computes the sum of the natural numbers up to n. In the IBP workflow, the programmer defines the situations before adding the transitions. An added transition is *consistent* iff the source situation together with the conditions and assignments on the arrow imply the target situation. A program is consistent iff all transitions are consistent. A program is *live* if at least one transition is enabled in each non-final situation. A program is *terminating* if each cycle decreases a variant (written in the upper right hand corner of the situation) which is bounded from below. Sound and complete proof rules for total correctness have been defined [4].

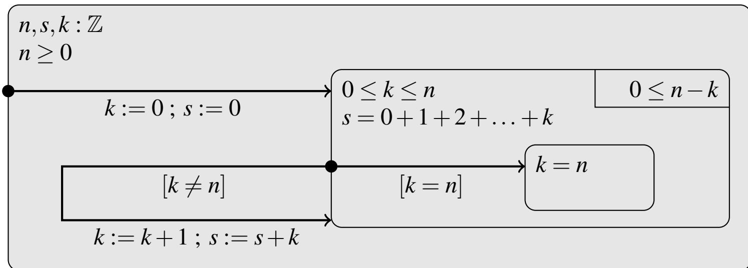


Fig. 1. An invariant diagram. Constraints (invariants) are written in the top left corner of situations, statements (guards and assignments) adjacent to transitions

PVS and Yices. PVS [13] is a verification system based on simply-typed higher-order logic. It has a rich specification language and supports predicate subtypes and dependent types. It includes an interactive proof system based on sequent calculus. Yices [9] is an SMT solver that supports uninterpreted functions with equality, linear real and

integer arithmetic, scalar types, recursive datatypes, tuples, records, extensional arrays, bit-vectors, and quantifiers. Yices is integrated into PVS; it can be invoked as a decision procedure with the command (yices), which either proves the current goal or does nothing.

Contribution. The paper comprises 1) a description of the Socos verification methodology, and 2) a case study illustrating how Socos together with PVS supports invariant based programming. For the first part, we introduce the Socos programming language and describe an embedding of it into PVS. The translation is based on the proof rules for IBP and weakest precondition calculus. Socos generates the consistency, liveness and termination conditions for a diagram, builds a proof script to decompose the conditions into fine grained lemmas, and then applies an *endgame strategy* to try to discharge each of them. For the case study, we verify an invariant based implementation of the heapsort algorithm. Heapsort is a reasonable exercise in verification. Firstly, although the code is small the verification involves a set of nontrivial invariants and proofs. Secondly, it is not trivial to write a correct implementation of heapsort. In particular, there is a corner case that is easily missed; we show how the tool aids in spotting such cases. Thirdly, the specification of heapsort involves the notion of permutation, which is infeasible to reason about in terms of the mathematical definition. The use of a background theory improves automation while maintaining soundness with respect to the definition.

Related work. PVS verification of Java programs is supported by Loop [14] and the Why/Krakatoa tool suite [10]. Several program verifiers are based on SMT solvers. Boogie [5] is an automatic verifier of BoogiePL, a language intended as a backend for encoding verification semantics of object oriented languages. Spec#, an extension to C#, is based on Boogie [6]. Back and Myreen have developed an automatic checker for invariant diagrams [3] based on the Simplify validity checker [8]. With the first author they later developed the checker into a prototype of the Socos environment [2].

Overview of paper. The sequel is organized as follows. Section 2 introduces the Socos language. Section 3 describes the verification methodology. Section 4 describes the translation of Socos programs into PVS. Section 5 shows our system in action as we build a verified implementation of heapsort. We conclude in section 6.

2 Socos Language

We introduce the Socos language in equivalent textual and diagrammatic notations using the following conventions. A production is written as $P ::= S$. The diagrammatic representation of a clause is shown to the right of the production. An optional part S of a clause is denoted $S^?$. Angle brackets group clauses, as in $\langle S\ T \rangle^?$. Repetition of S one or more times is denoted S^+; repetition zero or more times is denoted S^*. If the repeated elements are to be separated by a symbol , we write $S^{,+}$ and $S^{,*}$, respectively. Alternatives are separated with a bar, as in $S \mid T$. The nonterminal *Id* stands for the subset of PVS identifiers that do not contain the underscore character.[1] The nonterminals *Expr*, *TypeExpr* and *Const* stand for the PVS expressions, type expressions, and constant declarations, respectively. Their syntax and semantics are described in [12].

[1] The underscore character is reserved for the translation into PVS.

Expressions are type-checked by PVS to ensure that they are well defined, and can incur TCCs which may require theorem proving to discharge.

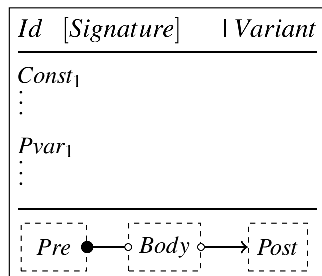
Contexts. The *verification context* is the top level program construct. It introduces a set of constants and a set of procedures:

$$
\begin{aligned}
\mathit{Context} ::= \ & \mathit{Id} : \textbf{context begin} \\
& \quad \langle \textbf{extending}\ \mathit{Id}^{+};\rangle^{?}\ \mathit{Importing}^{*}\ \langle \textbf{strategy}\ \langle \mathit{Id}|\mathit{String}\rangle;\rangle^{?} \\
& \quad \mathit{Const}^{*}\ \mathit{Procedure}^{*} \\
& \textbf{end}\ \mathit{Id}
\end{aligned}
$$

A context may *extend* one or more other contexts, inheriting their constants and procedures. The *Importing* clause is as in PVS and imports a background theory. The **strategy** clause defines the default proof command to be applied to the VCs. The argument is either one of the (transitively) extended contexts, in which case the strategy of that context is inherited; or it a PVS proof command. If omitted, the strategy is inherited from the first extended context, or, if the context is not an extension, defaults to (skip).

Procedures. A procedure is a program unit with a signature, pre- and postcondition specification, and implementation body:

$$
\begin{aligned}
\mathit{Procedure} ::= \ & \mathit{Id}\ [\mathit{Signature}] : \textbf{procedure} \\
& \mathit{Pre}^{?}\ \mathit{Post}^{?} \\
& \langle {**}\ \mathit{Variant}\ ;\rangle^{?} \\
& \textbf{begin} \\
& \quad \mathit{Const}^{*} \\
& \quad \mathit{Pvar}^{*} \\
& \quad \mathit{Body} \\
& \textbf{end}\ \mathit{Id}
\end{aligned}
$$



The signature consists of the lists of formal constant, value-result and result parameters:

$$
\begin{aligned}
\mathit{Signature} ::= \ & \langle\ \langle \mathit{Id}{:}\mathit{TypeExpr}\rangle^{,+}\rangle^{?} \\
& \langle \textbf{valres}\ \langle \mathit{Id}{:}\mathit{TypeExpr}\rangle^{,+}\rangle^{?} \\
& \langle \textbf{result}\ \langle \mathit{Id}{:}\mathit{TypeExpr}\rangle^{,+}\ \rangle^{?}
\end{aligned}
$$

Parameters have the following operational interpretations in a call. For constant parameters, the actuals are evaluated and bound to the corresponding formals; these bindings remain unchanged while the procedure is executed. For value-result parameters, the values of the actuals are copied to the formals, and when the procedure returns the final values of the formals are copied back to the actuals. Result parameters are used for passing a store for the result without regard to the initial value of the actual. The specification segment consists of the pre- and postconditions and an optional variant:

$$\mathit{Pre} ::= \langle \textbf{pre}\ \mathit{Expr};\rangle^{+} \qquad \boxed{\mathit{Expr}_1 \ \vdots} \qquad \Big|\ \mathit{Post} ::= \langle \textbf{post}\ \mathit{Expr};\rangle^{+} \qquad \boxed{\mathit{Expr}_1 \ \vdots}$$

$$\mathit{Variant} ::= \mathit{Expr}$$

The precondition of the procedure is the conjunction of the given (Boolean) expressions over the constant and value-result formals; the postcondition is the conjunction of the

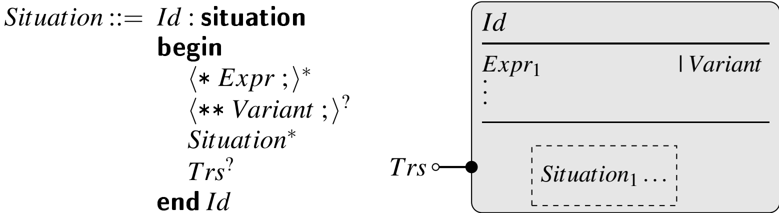
given predicates over the formals and initial-value constants for the value-result parameters. An omitted pre- or postcondition defaults to `true`. In diagrams, preconditions are drawn with a thick edge, postconditions with a double edge. If either is omitted, transitions are drawn to the edge of the procedure. A variant over the constant and value-result parameters must be given for (mutually) recursive procedures. The procedure body contains a set of local constants and local variables. The **pvar** keyword introduces one or more program variables of the same type:

$$\mathit{Pvar} ::= \textbf{pvar}\ \mathit{Id}^{,+} : \mathit{TypeExpr}\ ;$$

A program variable declaration has a strictly different interpretation than a PVS **var** declaration. The former adds a component to the state vector, whereas the latter binds a name to a type. The procedure body is an invariant diagram containing a (possible empty) set of *situations* and an (optional) initial *transition tree*:

$$\mathit{Body} ::= \mathit{Situation}^{*}\ \mathit{Trs}^{?}$$

Situations. A situation is a possibly empty sequence of *constraints*, an optional variant, a possibly empty sequence of nested situations, and an optional transition tree:



A situation declaration introduces a named predicate over the program variables. The predicate of a top level situation is the conjunction of its sequence of constraints; the predicate of a nested situation is the conjunction of the listed constraints and the predicate of the parent.

Transition trees. A transition tree is a tree where each leaf is a **goto** statement, each inner node is either a **choice** or an **if** command, and each edge may be labeled with a statement S:

$$\mathit{Trs} ::= \langle S\ ;\rangle^{?}\ \mathit{Tail}$$

$$\mathit{Tail} ::= \langle \mathit{Goto} | \mathit{Choice} | \mathit{If} \rangle$$

$$\mathit{Choice} ::= \textbf{choice}\ \mathit{Trs}^{+}\ \textbf{endchoice}$$

$$\mathit{Goto} ::= \textbf{goto}\ \mathit{Id} \ |\ \textbf{decreasing goto}\ \mathit{Id}$$

$$\mathit{If} ::= \textbf{if}\ \langle [\mathit{Expr}]\ ;\ \mathit{Trs}\rangle^{+}\ \textbf{endif}$$

A transition is described by the path from the root to a **goto** statement. The keyword **decreasing** declares that a transition decreases the variant of the target situation; this mechanism is used to define cutpoints for proving termination and is described in the next section. In diagrams, **decreasing** is marked with a double arrowhead. **choice** is nondeterministic choice, while **if** is guarded command. The former nondeterministically picks one of the branches for execution, while the latter nondeterministically picks an enabled branch for execution and fails if all guards are false.

Statements. The available statements are *assume*, *assert*, *assignment*, *havoc*, *sequential composition* and *procedure call*:

$$S ::= [Expr] \mid \{Expr\} \mid Id^{,+} := Expr^{,+} \mid Id^{,+} := ? \mid S_1 ; S_2 \mid Id(\langle Expr^{,+}\rangle^{?})$$

An assumption $[B]$ is executable only from states satisfying predicate B, from which it succeeds without effect on the state. An assertion $\{B\}$ fails if B is false; otherwise it succeeds without effect. An assignment statement $V_1,\ldots,V_n := E_1,\ldots,E_n$ evaluates in the current state the list of values $E_1,\ldots,E_n$ and assigns it to the variables $V_1,\ldots,V_n$. A havoc statement $V_1,\ldots,V_n :=?$ assigns nondeterministically to each V_i an arbitrary value from the type of V_i. Sequential composition executes the first statement followed by the second statement. Call invokes the specified procedure on the given actual parameters.[2]

3 Verification Methodology

An invariant diagram is correct if it is consistent, live and terminating [1]. Consistency means that each transition does not fail, and exits in a state satisfying the target situation. Liveness means that execution of the diagram does not stop before reaching the postcondition. Termination means that there are no infinite execution paths.

Consistency. A transition tree Trs_X from situation X with predicate J_X is consistent if $\mathsf{J}_X \subseteq [\![Trs_X]\!]$, where $[\![Trs_X]\!]$ denotes the *consistency condition* for the tree. Socos uses *weakest preconditions* as basis for generating consistency conditions. For the transition to target situation Y with predicate J_Y we have:

$$[\![S;\, \textbf{goto}\, Y]\!] = \mathrm{wp}(S)(\mathsf{J}_Y)$$

where $\mathrm{wp}(S)(\mathsf{J}_Y)$ is the largest set of states from which statement S must terminate in a state satisfying J_Y. A transition tree is consistent if all branches are consistent:

$$[\![S;\, \textbf{choice}\, Trs_1 \ldots Trs_n\, \textbf{endchoice}]\!] = \mathrm{wp}(S)([\![Trs_1]\!] \cap \ldots \cap [\![Trs_n]\!])$$

Procedure calls are verified based on the contract of the called procedure in the standard way.

Liveness. A procedure is live if: 1) there is an initial transition tree, and from every situation that is reachable from the initial transition tree at least one of the postconditions is reachable; 2) for every reachable transition, each command in the transition is able to proceed from any state in which it may be executed. Condition (1) is checked by reachability analysis of the procedure body. Condition (2) is ensured by the absence of assume statements.[3] Lack of liveness is not necessarily an error but rather an incompleteness in the program, so programs which are not live may still be verified for consistency. Incompleteness may be deliberate, e.g., as a stepping stone towards a final, live program. When liveness is desired, **if** rather than **choice** should enclose guarded transitions. It has the consistency condition

$$[\![S; \textbf{if}\, [G_1]; Trs_1 \ldots [G_n]; Trs_n\, \textbf{endif}]\!] = [\![S; \{G_1 \vee \ldots \vee G_n\}; \textbf{choice}\, [G_1]; Trs_1 \ldots [G_n]; Trs_n\, \textbf{endchoice}]\!]$$

and is thus consistent only if at least one guard is always enabled.

[2] For each constant formal, an expression of the same type or a subtype must be supplied. For each value-result formal, a program variable of the same type must be supplied. For each result formal, a program variable of the same type or a supertype must be supplied.

[3] All statements except assume satisfy the "excluded miracle" law: $\mathrm{wp}(S)(\emptyset) = \emptyset$.

Termination. Nontermination can be due to either unbounded iteration through cycles in the transition graph, or infinitely recursive procedure calls. Ensuring termination in the first case requires proving that no situation can occur infinitely often; i.e., that the transition relation is well-founded. Socos handles termination as follows. 1) The program is decomposed into its strongly connected components. 2) For each component, a situation X is picked such that each cycle back to X is cut by a transition marked as **decreasing**; an assertion $\{0 \leq \mathsf{V}_X < \mathsf{V0}_X\}$, where V_X is the variant of situation X and $\mathsf{V0}_X$ is a ghost variable storing the value of the variant at the root of the transition tree, is added to the end of each such transition; and an assertion $\{0 \leq \mathsf{V}_X \leq \mathsf{V0}_X\}$ is added to the end of each other transition in the cycle. 3) All transitions marked as **decreasing** are pruned, after which step (1) is applied recursively. Methodologically, termination verification is handled similarly to liveness in that it is optional. If the program graph cannot be reduced fully using this algorithm, Socos warns that the program may not be terminating. Termination verification of (mutually) recursive procedures is made simple: all mutually recursive procedures must share a variant, which must be shown to have decreased just before the recursive call.

4 Verifying Invariant Based Programs in PVS

During verification Socos generates a hierarchy of unparametrized PVS theories as follows. For each context C, a theory Ctx_C containing the importing and constant declarations of C is generated. For each extended context C', the theory $Ctx_{C'}$, is also imported. For each procedure P the theories $Spec_P$ and $Impl_P$ are generated. $Spec_P$ imports Ctx_C and contains the signature and pre- and postconditions of P. $Impl_P$ imports $Spec_P$ and for each called procedure P' imports $Spec_{P'}$; additionally it contains a PVS encoding of the state vector, the situation predicates, and the VCs for all transition trees. Generated identifiers include the underscore character to not clash with user defined names. Figure 2 shows an example translation: the left hand side shows a program consisting of two contexts, of which one contains a mutual recursion; the right hand side shows the generated PVS theories and the dependency (importing) relation.

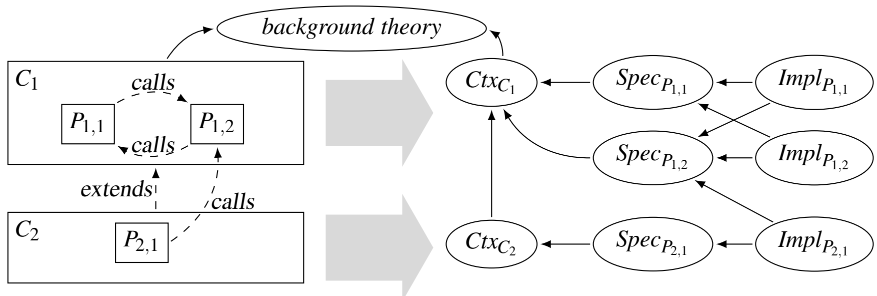


Fig. 2. Example PVS translation of two contexts. Contexts and procedures are drawn as boxes, theories as ellipses, calls and extends relations as dashed arrows, and importing relations as continuous arrows.

Each transition tree generates a VC consisting of two components: a lemma declaration, and an associated proof script to decompose the proof into separate goals for each constraint in the target situation of a transition and to apply the default proof strategy. The generated pair for situation X in the context of the state vector σ is:

$$\begin{array}{l}\mathsf{vc_}X : \mathsf{lemma\ forall}\ (\sigma) : \mathsf{J}_X(\sigma) \Rightarrow [\![[\![Trs_X]\!](\sigma)]\!]^{\mathsf{pvs}} \\ \%|\text{-}\mathsf{vc_}X : \mathsf{proof\ (skolem\text{-}2)\ (flatten\text{-}disjunct} + 1)\ [\![[\![Trs_X]\!](\sigma)]\!]^{\mathsf{prf}}\ \mathsf{qed}\end{array}$$

$[\![[\![Trs_X]\!](\sigma)]\!]^{\mathsf{pvs}}$ denotes the PVS rendition of the correctness condition $[\![Trs_X]\!]$ applied to σ. The second line is a ProofLite [11] script.[4] The command skolem-2 eliminates the top level quantifier.[5] The command flatten-disjunct then flattens the sequent so that the antecedent of the implication becomes the antecedent of the sequent; at this point the sequent is of the form $\alpha \vdash \beta$, where α is the precondition and β is the consistency condition of the transition tree. $[\![[\![Trs_X]\!](\sigma)]\!]^{\mathsf{prf}}$ denotes a proof script decomposing β into a subgoal for each constraint. Both translations are defined over the structure of the consistency condition. For a situation predicate, we have:

$$\begin{array}{l}[\![\mathsf{J}_X(\sigma)]\!]^{\mathsf{pvs}} = \mathsf{J}_X(\sigma) \\ [\![\mathsf{J}_X(\sigma)]\!]^{\mathsf{prf}} = (\mathsf{expand\text{-}defs}\ \mathit{defs\text{-}sexp})\ (\mathsf{check\text{-}report}\ \mathit{strat\text{-}sexp})\end{array}$$

The parameters *strat-sexp* and *defs-sexp* are S-expressions standing respectively for the default strategy to be applied to the leaves of the proof tree, and a listing of the definitions relevant to the sequent. The proof commands are Socos specific. expand-defs expands all situation predicates and splits the sequent into a separate branch for each situation constraint. Consequently, each proof goal produced by expand-defs is of the form $\alpha_1, \ldots, \alpha_n \vdash \beta$, where β is a situation constraint or an assertion, and each formula α_i originates from a source situation constraint, an assume statement, or an assertion. check-report applies *strat-sexp* to each goal, and then prints the status of the proof. The remainder of the translation is as follows:

$$\begin{array}{rcl}[\![(Q_1 \cap \ldots \cap Q_n)(\sigma)]\!]^{\mathsf{pvs}} &=& ([\![Q_1(\sigma)]\!]^{\mathsf{pvs}})\ \mathsf{and}\ \ldots\ \mathsf{and}\ ([\![Q_n(\sigma)]\!]^{\mathsf{pvs}}) \\ [\![(Q_1 \cap \ldots \cap Q_n)(\sigma)]\!]^{\mathsf{prf}} &=& (\mathsf{branch\ (split\text{-}n}\ n) \\ && \quad ((\mathsf{then}\ [\![Q_1(\sigma)]\!]^{\mathsf{prf}})\ \ldots (\mathsf{then}\ [\![Q_n(\sigma)]\!]^{\mathsf{prf}})))\end{array}$$

$$\begin{array}{rcl}[\![\mathrm{wp}(S_1;S_2)(Q)(\sigma)]\!]^{\mathsf{pvs}} &=& [\![\mathrm{wp}(S_1)(\mathrm{wp}(S_2)(Q))(\sigma)]\!]^{\mathsf{pvs}} \\ [\![\mathrm{wp}(S_1;S_2)(Q)(\sigma)]\!]^{\mathsf{prf}} &=& [\![\mathrm{wp}(S_1)(\mathrm{wp}(S_2)(Q))(\sigma)]\!]^{\mathsf{prf}}\end{array}$$

$$\begin{array}{rcl}[\![\mathrm{wp}([E])(Q)(\sigma)]\!]^{\mathsf{pvs}} &=& (E) \Rightarrow ([\![Q(\sigma)]\!]^{\mathsf{pvs}}) \\ [\![\mathrm{wp}([E])(Q)(\sigma)]\!]^{\mathsf{prf}} &=& (\mathsf{flatten\text{-}disjunct} + 1)\ [\![Q(\sigma)]\!]^{\mathsf{prf}}\end{array}$$

$$\begin{array}{rcl}[\![\mathrm{wp}(\{E\})(Q)(\sigma)]\!]^{\mathsf{pvs}} &=& (E)\ \mathsf{and}\ ([\![Q(\sigma)]\!]^{\mathsf{pvs}}) \\ [\![\mathrm{wp}(\{E\})(Q)(\sigma)]\!]^{\mathsf{prf}} &=& (\mathsf{branch\ (split\text{-}assert)} \\ && \quad ((\mathsf{then}\ (\mathsf{expand\text{-}defs}\ \mathit{defs\text{-}sexp}) \\ && \qquad\qquad (\mathsf{check\text{-}report}\ \mathit{strat\text{-}sexp})) \\ && \quad\ (\mathsf{then}\ [\![Q(\sigma)]\!]^{\mathsf{prf}})))\end{array}$$

[4] ProofLite is a PVS add-on by César Muñoz that allows embedding PVS proofs (as S-expressions) in the main theory file rather than in a separate `.prf` file.

[5] skolem-2 is an adaptation of skolem. Instead of generating new names for the Skolem constants, it reuses the names from the binding expressions; this makes the VC slightly easier to read.

$$[\![\mathrm{wp}(X := E)(Q)(\sigma)]\!]^{\mathsf{pvs}} = (\mathsf{lambda}\ (X) : [\![Q(\sigma)]\!]^{\mathsf{pvs}})\ (E)$$
$$[\![\mathrm{wp}(X := E)(Q)(\sigma)]\!]^{\mathsf{prf}} = (\mathsf{beta}\ +)\ [\![Q(\sigma)]\!]^{\mathsf{prf}}$$

$$[\![\mathrm{wp}(X := ?)(Q)(\sigma)]\!]^{\mathsf{pvs}} = \mathsf{forall}\ (X) : [\![Q(\sigma)]\!]^{\mathsf{pvs}}$$
$$[\![\mathrm{wp}(X := ?)(Q)(\sigma)]\!]^{\mathsf{prf}} = (\mathsf{skolem\text{-}2})\ [\![Q(\sigma)]\!]^{\mathsf{prf}}$$

split-n and split-assert are Socos specific strategies. The command split-n is an iterated version of split: it splits the top-level conjunction into two branches, and then repeatedly splits the right branch for a total of at most $n-1$ iterations. Hence, a sequent of the form $\Gamma \vdash \alpha_1 \wedge (\alpha_2 \wedge (\alpha_3 \wedge \ldots))$ is split into the goals $\Gamma \vdash \alpha_1, \Gamma \vdash \alpha_2, \Gamma \vdash \alpha_3 \ldots,$. The command split-assert splits a sequent of the form $\Gamma \vdash \alpha \wedge \beta$ into the two goals $\Gamma \vdash \alpha$ and $\Gamma, \alpha \vdash \beta$.

In general, any PVS proof strategy can be used as the default strategy. In our case study we use the following to try to discharge VCs automatically:

```
(defstep endgame (&optional (lemmas nil))
    (let ((introduce-lemmas `(then ,@(loop for l in lemmas append `((lemma ,l))))))
        (then
            (skosimp*)
            (auto-rewrite-defs :always? t)
            (assert)
            introduce-lemmas
            (yices)
            (fail)))
    "End-game strategy" "Invoking Yices, supplying lemmas: ~{~a~^, ~}")
```

The above strategy expands all relevant definitions in the sequent, loads the supplied lemmas into the antecedent, and invokes Yices. Yices either proves the lemma, or the entire strategy fails. Definitions not expanded in the second step appear as uninterpreted constants and the supplied lemmas as axioms to Yices. This mechanism allows supplying specific properties in cases where reasoning in terms of the definition is infeasible; we will use this technique in practice in the next section.

5 Heapsort: An Exercise in PVS Supported Invariant Based Programming

Heapsort is an in-place, comparison based sorting algorithm from the class of selection sorts. It achieves $O(n \log n)$ worst and average case time complexity by the use of a binary max-heap, which allows constant time retrieval of the maximal unsorted element and logarithmic time restoration of the heap property. By storing the heap in the unsorted portion of the array, heapsort uses only a small constant amount of additional memory. The algorithm implemented here is close to the one given by Cormen et al. in [7, Ch. 6]. It comprises two procedures. The main procedure, heapsort, first builds a max-heap out of an unordered array, and then deconstructs the heap one element at a time, each iteration extending the sorted portion by swapping the root of the heap with the last element of the heap. Restoring the heap property after swapping is accomplished by an auxiliary procedure called siftdown.

Background theory. In Socos the type vector is defined as a parametric record type [#len : nat, elem : [{i : nat|i < len}->T]#], where T is the element type. For vector a we have the abbreviations a(i) for elem(a)(i), and index(a) for the type {i : nat|i < len(a)}. We create a new background theory called sorting (in the sequel we use sorting as a background theory, extending it with additional definitions as needed) and introduce a predicate sorted to express that an integer array is sorted in non-decreasing order:

```
sorting: theory
begin
    a,b,c: var vector[int]
    sorted(a):bool=forall (i,j:index(a)):i<j=>a(i)<=a(j)
    ...
```

To express that a sorted array preserves the elements of the original array, we introduce a binary relation perm over vectors:

```
perm(a,b):bool=
    exists (f:(bijective?(index(a),index(b)))):forall (i:index(b)):b(i)=a(f(i))
```

While this notion of permutation is mathematically concise, reasoning in terms of this definition gives little hope for automation as it requires demonstration of a bijection. In our case it is more fruitful to consider permutation as the smallest equivalence relation that is invariant under pairwise swapping. We add these four lemmas to the background theory and prove them in PVS:

```
perm_len: lemma perm(a,b) => len(a)=len(b)
perm_ref: lemma perm(a,a)
perm_sym: lemma perm(a,b) => perm(b,a)
perm_trs: lemma perm(a,b) and perm(b,c) => perm(a,c)
```

The first lemma allows us to infer that permutations have equal length, whereas the remaining three state that permutation is an equivalence relation. We also introduce a function swap for exchanging the elements at indexes i and j, while keeping the remainder of the elements in the array unchanged:

```
swap(a,(i,j:index(a))): {b|len(b)=len(a)} = a with [i:=a(j),j:=a(i)]
```

As all array manipulations will be through swap, the prover only needs to know the effect of swap on subsequent array reads, and that swap maintains permutation. We add and prove the following (trivial) lemmas:

```
swap_acc: lemma forall(a,(i,j,k:index(a))):
                    swap(a,i,j)(k) = a( if k=i then j elsif k=j then i else k endif )

swap_perm: lemma forall(a,(i,j:index(a))):perm(a,swap(a,i,j))
```

Finally, we add the declaration:

```
auto_rewrite- perm,swap
```

This prevents perm and swap from being expanded into their definitions, and hence they will be handled as uninterpreted functions by Yices when the endgame strategy is invoked.

The heapsort **procedure.** Next, we introduce a new context heapsort which imports the background theory and applies endgame with the six properties defined so far. The rest of the context heapsort will be the two procedures, which will be presented as invariant diagrams.

```
heapsort : context
begin
  importing sorting;
  strategy "(endgame :lemmas (perm_len perm_ref perm_sym perm_trs
                               swap_acc swap_perm))";
  ⟨ implementation of heapsort ... ⟩
  ⟨ implementation of siftdown ... ⟩
end heapsort
```

The procedure heapsort, given a value-result parameter a of type vector[int], should establish the postcondition $\mathsf{sorted}(a) \wedge \mathsf{perm}(a, a_0)$, where a_0 is the value of a at the start of the execution of the procedure. We design heapsort around the situations BUILDHEAP and TEARHEAP. BUILDHEAP constructs the heap out of the unordered array by moving in each iteration one element of the non-heap portion of a into its correct place in the heap portion; TEARHEAP then sorts a by selecting in each iteration the first (root) element from the heap portion and prepending it to the sorted portion of the array. We use a loop variable k ranging over the interval $[0..\mathsf{len}(a)]$ in both both situations. The invariant $\mathsf{perm}(a, a_0)$ is also to be maintained throughout both situations.

In BUILDHEAP, the heap is extended leftwards one element at a time by decreasing k. The portion to the right of k satisfies the *max-heap property*: an element at index i is greater than or equal to both the element at index $2i+1$ (the left child) and the element at index $2i+2$ (the right child). For each iteration, after k has been decremented the new element at position k must be "sifted down" into the heap to re-establish the max-heap property. We use a separate procedure, siftdown, for this purpose. The parameters to siftdown are the left and right bounds of the heap, as well as the array itself. We implement and verify siftdown in the next section.

We now formalize the heap property in the background theory. We add to the sorting theory the functions l and r for the index of the left and right child, respectively, as well as a predicate heap expressing that a subrange of an array satisfies the max-heap property (note that the operators or, and are conditional in PVS):

```
l(i:nat):nat=2*i+1; r(i:nat):nat=2*i+2

heap(a,(m,n:nat)):bool=
    m<=n and n<=len(a) and
    (forall (i:nat): m<=i =>
        (l(i)<n=>a(i)>=a(l(i))) and (r(i)<n=>a(i)>=a(r(i))))
```

We then have that BUILDHEAP should maintain $\mathsf{heap}(a, k, \mathsf{len}(a))$. Upon termination of the loop, $\mathsf{heap}(a, 0, \mathsf{len}(a))$ should hold.

In the situation TEARHEAP we again iterate leftwards, this time maintaining a heap to the left of k, and a sorted subarray to the right of k. The loop is iterated while k is greater than one—when it is smaller than or equal to one the array is sorted. In each iteration, k is first decremented, then the element at index k element is exchanged with

the element at index 0 (the root of the heap) to extend the sorted portion. Since the leftmost portion may no longer be a heap, this is followed by a call to siftdown starting from index 0 to restore the heap property. Additionally, to infer that the extended right portion is sorted, we also need to know that the array is *partitioned* around k, i.e., that the elements to the left of k are smaller than or equal to the elements to the right of (and at) k.

To express the constraints of TEARHEAP we introduce two definitions; that the right-most portion of an array is sorted, and that an array is partitioned at a given index:

```
sorted(a,(n:upto(len(a)))):bool= forall(i,j:index(a)):n<=i and i<j => a(i)<=a(j)

partitioned(a,(n:upto(len(a)))):bool= forall(i,j:index(a)):i<n and n<=j=>a(i)<=a(j)
```

TEARHEAP should then maintain $\mathsf{partitioned}(a,k) \wedge \mathsf{sorted}(a,k) \wedge \mathsf{heap}(a,0,k)$.

We proceed by adding the initial transition, the final transitions, and the transition between BUILDHEAP and TEARHEAP. Figure 3 shows the diagram with these transitions in place. For the initial transition one possibility is to initialize the loop counter to len(a). However, we can do better by noting that heap(a, m, n) is true for any m and n such that $\lfloor \mathsf{len}(a)/2 \rfloor \leq m \leq n \leq \mathsf{len}(a)$, i.e. the right half of the unsorted array already satisfies the max-heap property (since its elements are leaves). We can directly try to confirm this hypothesis by asking Socos to check the partial implementation. The tool responds that all transitions are indeed consistent, but since the diagram is incomplete, it also points out that the procedure may not be live. However, before adding the loop transitions we need to actually implement and verify the siftdown procedure.

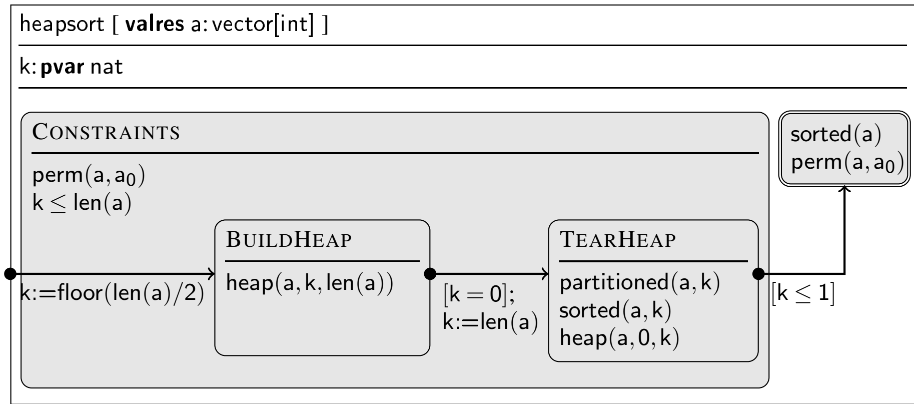


Fig. 3. Partial implementation of heapsort

The siftdown procedure. The parameters to the siftdown procedure are the left bound m, the right bound n, and the array a. Assuming the subrange $a[m+1..n)$ satisfies the heap property, siftdown should ensure upon completion that the subrange $a[m..n)$ satisfies the heap property, that the subranges $a[0..m)$ and $a[n..\mathsf{len}(a))$ are unchanged, and that the updated array is a permutation of the original array. siftdown achieves its

postcondition by moving the first element in the range downward into the heap until it is greater than or equal to both its left and right child, or the bottom of the heap has been reached. When either condition is true, the heap property has been restored. Each iteration of the loop swaps the current element with the greater of its children, maintaining the invariant that each element within the heap range, except the current one, is greater than or equal to both its children. Consequently, the invariant must include that the parent of the current element is greater than or equal to the children of the current element.

The loop terminates when either the values of both children are less than or equal the current element, or there are no more children within the range of the heap; i.e., when the condition $\mathsf{n} \le \mathsf{r}(\mathsf{k}) \vee (\mathsf{a}(\mathsf{l}(\mathsf{k})) \le \mathsf{a}(\mathsf{k}) \wedge \mathsf{a}(\mathsf{r}(\mathsf{k})) \le \mathsf{a}(\mathsf{k}))$ becomes true. Figure 4 shows a diagram with an intermediate situation SIFT and the entry, loop and exit transitions in place. The predicate $\mathsf{eql}(a,b,i,j)$ is an abbreviation for the property $(\forall r : i \le r < j \Rightarrow a(r) = b(r))$. We have used guarded commands (solid square) to enforce liveness conditions, and we have also provided the variant $\mathsf{n} - \mathsf{k}$ for situation SIFT and marked both loop transitions as decreasing. If we try to verify the program, Socos informs us that all transitions except the exit transition could be proved. Our tool is unable to prove that $\mathsf{heap}(\mathsf{a},\mathsf{m},\mathsf{n})$ is established by the exit transition. By more careful inspection it is easy to see that the condition is actually not provable due to the omission of a corner case in the program in Figure 4. The bug is that when $\mathsf{n} = \mathsf{r}(\mathsf{k})$, nothing is known about the relation between $\mathsf{a}(\mathsf{k})$ and $\mathsf{a}(\mathsf{l}(\mathsf{k}))$. This case occurs when the left child of the current element is the last element in the heap range, and the right child falls just outside of the

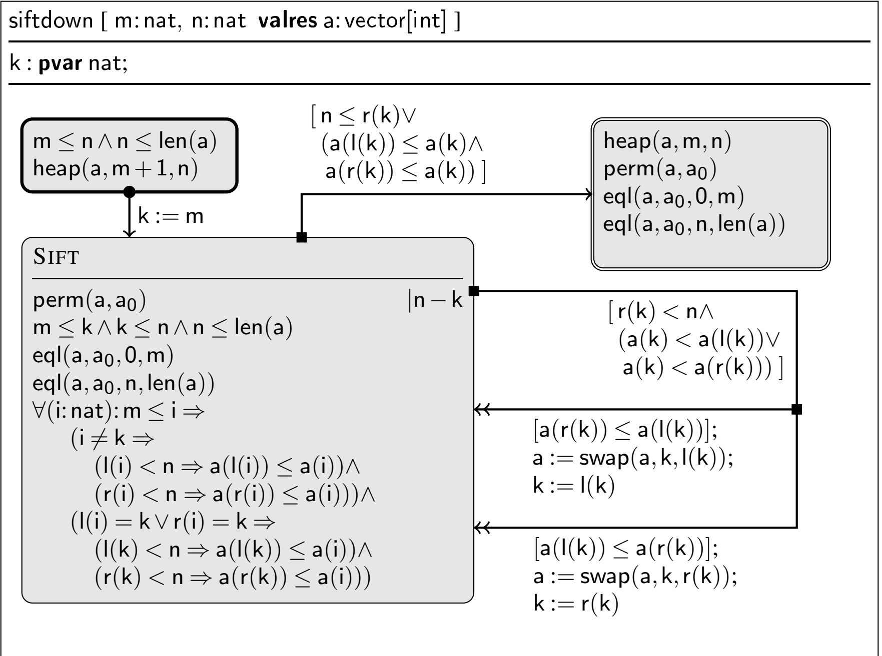


Fig. 4. First attempt at siftdown

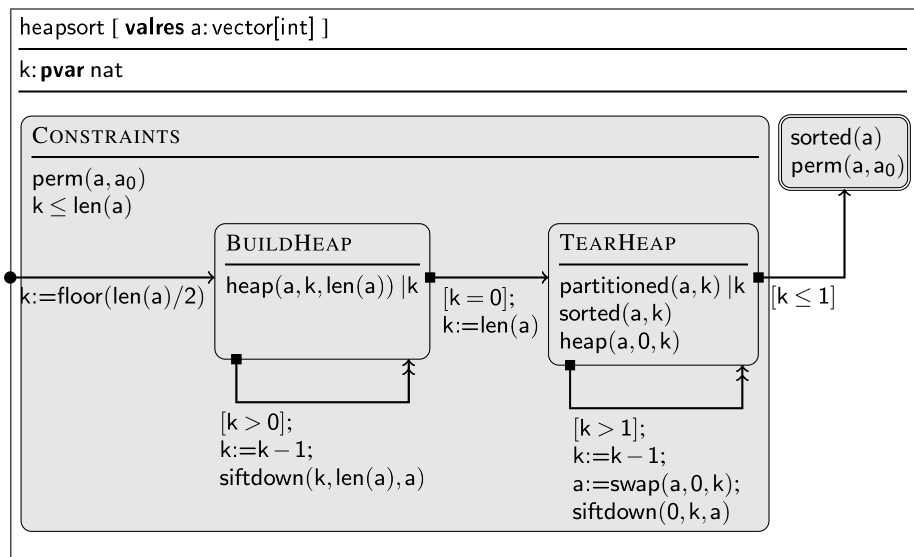


Fig. 5. Complete implementation of heapsort

heap range. To confirm this guess, we can strengthen the first disjunct of the exit guard to $n < r(k)$ and re-check. Now, the exit transition is proved consistent, but the liveness assertion for the first branch from SIFT now fails since the case $n = r(k)$ is no longer handled. We can resolve the issue by restoring the first disjunct of the exit guard to $n \leq r(k)$, and instead handling the corner case in a separate branch of the exit transition which swaps elements k and $l(k)$ if $a(k) < a(l(k))$ before exiting to the postcondition. After this correction all the VCs are discharged automatically.

Completing heapsort**.** Using siftdown we can now complete heapsort. Figure 5 shows the program in Figure 4 with loop transitions, procedure calls and variants added. If we run the program in Figure 5 through the verifier, all termination and liveness conditions are discharged automatically. The consistency conditions are also all proved except one:

```
procedure 'heapsort', constraint(s) in transition from
'TearHeap' to 'TearHeap':
    {-1}  0 <= k - 1
    {-2}  k - 1 < k
    {-3}  (heap(a_1, 0, k - 1))
    {-4}  (perm(a_1, swap(a, 0, k - 1)))
    {-5}  (eql(a_1, swap(a, 0, k - 1), 0, 0))
    {-6}  (eql(a_1, swap(a, 0, k - 1), k - 1, len(a_1)))
    {-7}  0 <= k - 1
    {-8}  k - 1 <= len(swap(a, 0, k - 1))
    {-9}  (heap(swap(a, 0, k - 1), 0 + 1, k - 1))
    {-10} k > 1
    {-11} ((k > 1 OR k <= 1))
    {-12} (perm(a, a_0))
```

```
{−13}  k <=  len(a)
{−14}  (partitioned(a,  k))
{−15}  (sorted(a,  k))
{−16}  (heap(a,  k))
   |-------
[1]    (partitioned(a_1,  k − 1))
```

In the above a_1 denotes the value of a returned by siftdown. Here the verifier is unable to prove that the procedure call maintains partitioned. This condition is actually true, but hard to prove because of the way we have defined the postcondition of siftdown. The procedure modifies the leftmost portion of the array, but the properties of perm sent to Yices cannot be used to infer that partitioned is maintained throughout the call. Proving the condition without modifying the specification of siftdown actually requires two additional properties: that the root of a max-heap is the maximal element; and that if partitioned holds for an index and an array, it also holds for a permutation of the array where the portion to the right of the index is unchanged. We can make the conditions explicit in the program by adding assertions to the TEARHEAP loop transition statement as follows:

```
[k>1] ; k:=k−1 ; {forall (i:index(a)):i<=k => a(i)<=a(0)} ;
a:=swap(a,0,k) ; {partitioned(a,k)} ; siftdown(0,k,a)
```

Re-checking the program, we are now left with two (simpler) conditions: the first assertion, and the same condition as above with the added assertions as additional assumptions (the second assertion is discharged automatically). The first condition requires an induction proof. Proving that $\mathsf{partitioned}(\mathsf{a_1}, \mathsf{k}-1)$ is a consequence of $\mathsf{partitioned}(\mathsf{swap}(\mathsf{a}, 0, \mathsf{k}-1), \mathsf{k}-1)$ is a bit more involved and requires reasoning in terms of the definition of permutation. To finish the verification, we prove and add the following two lemmas to the background theory:

```
heap_max: lemma forall (k:nat): heap(a,0,k) => forall (i:nat): i<k => a(i)<=a(0)

perm_partitioned: lemma forall (a,b,(k:upto(len(a)))):
                      perm(a,b) and partitioned(a,k) and eql(a,b,k,len(a))
                      => partitioned(b,k)
```

With the aid of these added properties, the goals are discharged automatically.

6 Conclusions

This paper presented a concrete invariant based programming language and its translation into PVS. The translation is implemented in the Socos verification tool. We have given an example of the Socos workflow by developing a verified implementation of heapsort. In the Socos workflow, the key lemmas are defined and proved interactively in background theories. In our experience, striving to make background theories general and implementation-independent enhances the verification experience and allows a library of reusable theories to accumulate. We have also shown that the tool also helps

in detecting bugs when verification fails; our experience is that with some practice it becomes easy to spot the mistakes in the program based on the unproven conditions.

IBP is a new technique and has not been applied in large scale case studies yet. PVS is, however, a venerable player in the formal methods field. Our experience is that PVS provides a natural, rich specification environment, and that Yices efficiently discharges most of the simple conditions. This work is a step towards scaling up invariant based programming by allowing large existing libraries of mathematical theories, such as the PVS prelude and the Nasa libraries, to be used.

Acknowledgments. We would like to thank Viorel Preoteasa for useful comments on the translation to PVS and for suggesting heapsort as a case study.

References

1. Back, R.-J.: Invariant based programming: Basic approach and teaching experience. Formal Aspects of Computing 21(3), 227–244 (2009)
2. Back, R.-J., Eriksson, J., Myreen, M.: Testing and verifying invariant based programs in the SOCOS environment. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 61–78. Springer, Heidelberg (2007)
3. Back, R.-J., Myreen, M.: Tool support for invariant based programming. In: Proc. of APSEC 2005, pp. 711–718. IEEE Computer Society, Los Alamitos (2005)
4. Back, R.-J., Preoteasa, V.: Semantics and proof rules of invariant based programs. Technical Report 903, TUCS, Turku, Finland (July 2008)
5. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
6. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
7. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education, New York (2001)
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. of the ACM 52(3), 365–473 (2005)
9. Dutertre, B., de Moura, L.: The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA (August 2006)
10. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-73368-3_21
11. Munõz, C.: Batch proving and proof scripting in PVS. Technical Report NASA CR-2007-214546 / NIA 2007-03, NASA Langley Research Center and National Institute of Aerospace (2007)
12. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference, (Version 2.4). Computer Science Laboratory, SRI International, Menlo Park, CA (November 2001)
13. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
14. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 299–312. Springer, Heidelberg (2001)

Proof Obligation Generation and Discharging for Recursive Definitions in VDM

Augusto Ribeiro and Peter Gorm Larsen

Aarhus School of Engineering, Aarhus University, Denmark
ari@iha.dk, pgl@iha.dk

Abstract. A proof obligation is a theorem stating that a certain property must hold in order for a formal specification to be internally consistent. If a proof obligation can be proved, then the referred part in the specification is consistent. The generation of proof obligations to check for a specification's internal consistency is a concept that has been applicable in a VDM context for a long time. This work is extending the existing proof obligation generation capabilities with proof obligations for the termination of recursive functions. Those proof obligations can then automatically be moved over to HOL and the corresponding proofs can be carried out in that framework. Depending upon the nature of the recursion, the discharge of these proofs can be done automatically. This paper will categorise the different kinds of recursion.

1 Introduction

VDM contains constructs such as type invariants defined as general predicates that make it impossible to create an automatic static check to determine whether all parts of a given VDM model are internally consistent. For all the places which potentially could yield inconsistencies such as a run-time errors, it is instead possible to precisely describe the semantic property that would guarantee consistency. These properties are termed "proof obligations" (POs) in a VDM context. The automatic production of such proof obligations for VDM has been documented in the literature [1]. However, none of the work in a VDM context has so far considered the proof obligations necessary to ensure the termination of recursive functions. This has naturally been done in other formalisms such as PVS [2] but many of these have not been able to cope with mutually recursive definitions as it is treated in this paper.

The tool for automatic production of POs is called a Proof Obligation Generator (POG) and this is itself developed using VDM. The POG produces POs for a variety of consistency conditions but in this paper we focus solely on the POs for the termination of recursive functions [3]. VDM POs can be automatically translated to HOL [4,5] where these can be formally verified. So rather than reinventing the wheel the strategy here is to reuse the powerful proof support in an existing theorem prover such as HOL [6]. In this paper we solely focus on the extensions necessary to generate, transform and subsequently prove the POs for recursive functions in VDM. VDM also contains operations that can access state but these are not covered in this paper.

Section 2 provides the basic definitions and terminology required to understand recursive definitions in the VDM context. This is followed in Section 3 by the proof obligations related to the different categories of recursive functions and their termination.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 40–55, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Section 4 presents extracts of the VDM model that have been made to extend the existing POG from VDMTools and Overture with POs for termination of recursive functions. Section 5 introduces the use of a theorem prover (HOL) to prove the generated proof obligations and the level of automation for different categories of recursive functions. Section 6 demonstrates more details about how HOL can be used for discharging the proof obligations generated for the different categories of termination POs. Finally Section 7 and Section 8 provide pointers to related work, concluding remarks and further work respectively.

2 Recursive Definitions and VDM

Recursion is an algorithmic technique where a function is defined in terms of itself for a "smaller" part of the task at hand. Such definitions are only well-founded if the recursion eventually reaches a base case without recursion. This is called termination from an operational point of view and this is a desirable property that we would like to be able to formally prove for all recursive functions that can be defined in VDM.

2.1 Termination

To prove termination of a functional program there has to be a well-founded ordering such that the arguments in each recursive call are smaller than the corresponding inputs. An ordering $\succ$ is well founded if there exists no infinite descending chain.

$$x_1 \succ x_2 \succ \ldots$$

Given that there are no infinite descending chains, one of the arguments of a recursive call is guaranteed to hit a case in which it will have to stop descending and thus the function terminates.

The set of the natural numbers, *nat*, forms a well-founded order under '$<$' so this is chosen as the default well-founded ordering.

In some cases there is the need to use a lexicographic ordering in order to prove termination. A lexicographic order is used to compare tuples.

$$(a, b) \succ (a', b') \text{ if and only if } a \succ a' \text{ or } (a = a' \text{ and } b \succ b')$$

Lexicographic orders can be extended to tuples of arbitrary length. Termination proof of some functions requires this kind of ordering denoted in VDM by the **LEXn** keyword (only used in proof obligations) where **n** is the number of elements in the tuple. The following statement is an example that would be valid when using a lexicographic order.

```
(1,10) (LEX2 <) (2,1) (LEX2 <) (2,10)
```

2.2 Measure

A measure is generally a function from the parameter data type of the recursive function to a given well founded ordering. The method for using arbitrary measures to prove termination of recursive functions was suggested in [7]. As a consequence of the chosen

order for proofs, all the measure functions in this work will have Nat as range. The measure function is needed for the generation of all POs so it will be supplied by the user for each recursive function definition using the new keyword **measure**. The function `fac` that computes the factorial of a number shows how this keyword is used:

```
fac : nat -> nat
fac(n) == if n = 0 then 1 else n*fac(n-1)
measure id
```

In which `id`, in this case, the identity function is the measure function defined as:

```
id : nat -> nat
id(n) == n
```

This example only illustrates the use of the **measure** keyword and `id` will be suitable here since $n - 1 < n$ will form a finite descending chain towards the base case for the natural numbers.

Note that it is required that the measure function is that the measure itself must be a function that terminates.

2.3 The Vienna Development Method

VDM is a well-established formal method that uses a few formal modelling languages, each supporting different forms of system specification. VDM++ [8] extends the ISO standardised VDM-SL [9] with features for object-oriented modelling and concurrency. The proof work presented here supports a functional subset of VDM that is common to both VDM-SL and VDM++ except that the proof obligations generated include POs that are strictly not necessary according to the standard logic for VDM which is called Logic of Partial Functions (LPF) [10]. The proof obligations generated from a tools perspective follow a standard left-to-right evaluation strategy [11] (known from programming languages) whereas LPF does not require this. Here *functions* represent mathematical functions and functions can be defined both implicitly and explicitly in VDM. Explicit functions can also have pre-conditions.

VDMTools [12,13] is an industrial quality tool that supports static type checking and the generation of POs. More recently the Overture open source initiative built on top of Eclipse has also incorporated such features for all VDM dialects [14]. They have a number of other features that are not important for the contents of this paper.

2.4 From VDM Models to Proved Proof Obligations

From a VDM model, one can generate the termination POs with VDMTools and Overture. If the generated POs contain references to any VDM model definitions, it means that these definitions also need to be translated to HOL in order to give meaning to those references in HOL.

When HOL is used to prove properties about a VDM model, it is necessary to define semantically equivalent functions in HOL for which termination must be proved. This essentially leads to redundant termination proofs because the POG for VDM also

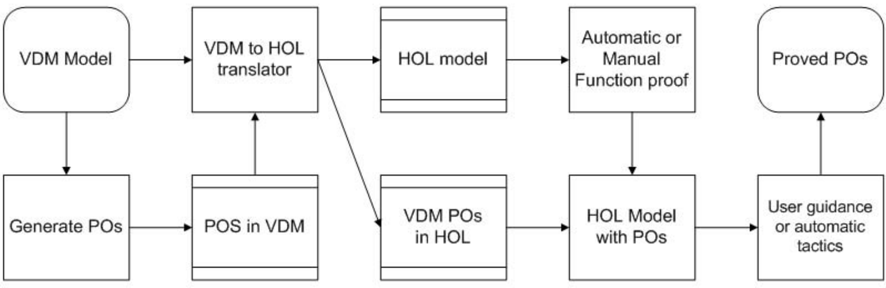


Fig. 1. From VDM models to proved POs

generates the same termination POs. This only happens if one needs to use the function definitions in HOL. POs that contain only references to the measure do not suffer from this problem, unless if the measure is itself recursive. Though some generated POs are redundant when using HOL, it is not the case if the user wishes to manually prove them or use another tool. In some cases HOL is able to prove them automatically, but manual proofs may be needed.

3 Generating Proof Obligations for Recursive Functions

3.1 POs in VDM

The current POG for VDM used in VDMTools was originally specified in [15]. The general form of a PO is the same as a proof rule. They contain the context information as assumptions and some property that must be proved as conclusion. In the next examples the POs will be presented in an ASCII representation like this:

```
PO: context information ==> predicate
```

where the context information typcally is provided as universal quantifications.

3.2 Simple Recursion

A recursive function has normally the following general form, *r* being the argument of the recursive call:

$$f(x) = \ldots f(r) \ldots$$

The factorial is an example of this type (see section 2.2). A proof obligation is generated using the context as assumption. The recursive call is in the *else* branch so the context information for it is '`not (n=0)`'. The statement that should be proved is created based on the argument of the function and the input for recursion. The PO for *fac* looks like:

```
forall n:nat & not(n=0) ==> id(n) > id(n-1)
```

Meaning that the recursive call argument should be smaller than the input of the function after the measure is applied on each of them. The PO contains an assumption that the context of the recursive call as mentioned above.

When a function has nested recursion, the generated PO is different. The nested recursion can have arbitrary levels but here is the most simple case where the function has only one nested recursive call.

$$f(x) = ...f(f(r))...$$

This type of function needs a different type of approach because the condition to ensure termination is that the argument of both calls to the function must be decreasing.

The following example demonstrates how to generate POs for a recursive nested function called *nest* that is defined in VDM++ as:

```
nest : nat -> nat
nest(n) == if n = 0 then 0 else nest(nest(n-1))
measure id;
```

The PO should contain conditions about both recursive calls of the function *nest*. The inner recursion yields the following PO.

```
forall n:nat & not(n=0) ==> id(n) > id(n-1)
```

For the outer recursion the PO is:

```
forall n:nat & not(n=0) ==> id(n) > id(nest(n-1))
```

These conditions state that the input for the recursive calls of nest should be decreasing for the innermost and outermost calls. So the final PO for *nest* is:

```
forall n:nat & not(n=0) ==> id(n) > id(n-1) and
                            id(n) > id(nest(n-1))
```

3.3 Mutual Recursion

A mutual recursive function definition is characterized by two or more functions whose algorithms are mutually dependent. This is, a function f calls another function g, which in turn calls f creating a mutual dependency.

$$f(n) = ...g(n')...$$
$$g(r) = ...f(r')...$$

POs for mutual recursive definitions only differ from the ones above because they involve a measure from each function present in the definition. Apart from that the process is the same. The functions presented below are mutual recursive. The measure for them is the presented below, one for each.

```
m_even : nat -> nat
m_even(n) == 2 * n + 1;

even: nat -> bool
even(n) ==
  if n = 0 then true
  else odd(n);
measure m_even;
```

```
m_odd : nat -> nat
m_odd(n) == 2 * n;

odd : nat -> bool
odd(n) ==
  if n = 0 then false
  else even(n-1)
measure m_odd;
```

The recursion in `even` yields the right side of the PO conjunction and `odd` the left side. They say that the input for the recursive call of `odd` (`even` respectively) should be smaller then the input of the `even` (`odd`) function respectively.

```
forall n:nat & not(n=0) => m_even(n) > m_odd(n)
and
forall n:nat & not(n=0) => m_odd(n) > m_even(n-1)
```

If the arguments are decreasing in each of the functions then the mutual recursive definition can be said as to be terminating.

3.4 Preconditions

Proof obligations for functions with a precondition require that they are recorded as an assumption. The relation between input and recursive argument needs to be valid only if they fulfill the precondition. The next function `fnpre` illustrates this:

```
fnpre : nat -> nat
fnpre(x) ==
  if x=3 then 3 else fnpre(x-1)
pre x >= 3
measure id;
```

For this function a predicate named '`pre_fnpre`' is implicitly defined that computes its precondition. This predicate will be used in the construction of the PO. The PO for `fnpre` is:

```
forall x:nat & pre_fnpre(x) => (not(x=3) => id(x) > id(x-1))
```

It is only required that the recursion relation is valid for the values satisfying the precondition. So functions with preconditions can be treated as every other, they just have the precondition added to the context of each recursive case.

4 Implementation of the Proof Obligation Generator

In this section the process in which proof obligations are extracted from a VDM specification is explained. The parts of VDMTools and Overture that needed to be changed in order to generate the proof obligations for recursive functions are also mentioned. In figure 2 several components that take part on the process are shown and also how they are connected.

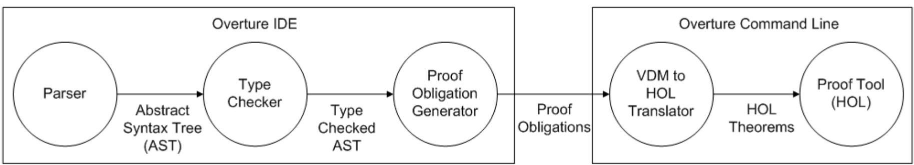


Fig. 2. Overture proof obligation data flow

4.1 VDMTools

The first step is to use the VDM Parser on a VDM model to get it represented as an equivalent AST (Abstract Syntax Tree). The Parser had to be changed to take into account of the new **`measure`** keyword and how it should be used in VDM. If the Parser is able to parse the VDM specification, the result will be an abstract syntax tree that is fed to the type checker.

It is in the Type Checker that the majority of the changes were made. It is responsible for determining which functions are recursive and which ones of them have a defined **`measure`** function. Here the domain data types of functions and their measures are checked for equality. Essentially this is carried out making a transitive closure of the function applications. Depending on the fulfillment of all the requirements, warnings or errors may be generated and presented in VDMTools and Overture GUI. A more thorough explanation of all warnings and errors is present in [3].

If the Type Checker completes without errors, it is possible to generate proof obligations. The information gathered during the type checking phase is reused by the Proof Obligation Generator.

4.2 Overture

In Overture, the detection of a recursive function f is only one step. f is marked as recursive if on any application of a function on f's body, the applied function matches the name of f. The detection of mutual recursive functions is a simple improvement that currently is in progress following the same principles used for VDMTools.

5 Proving POs for Recursive Functions in HOL

In regard to proving the proof obligations, different patterns of recursion in functions can be distinguished. Each one of them requires a different proof approach.

5.1 Straight Forward Recursion

Functions with straightforward recursion patterns are normally simple to prove.

$$f(x) = ...f(r)...$$

This is the class that HOL easily proves automatically too. The proof normally consists of simple rewrites of the function definition and application of arithmetic rules.

5.2 Nested Recursion with Nested Function Different from the Calling One

Here we consider functions with nested recursion patterns, with calls to another recursive function inside the recursive call.

$$f(x) = ...f(...g(r)...)...$$

The PO generated for this type of recursive function involves proving that the argument of the call after being transformed by the other function is still "smaller" than the input.

The strategy presented here to tackle the termination proof of this kind of function was first suggested in [16]. The `log` example will be used to ilustrate the approach to this kind of recursion. The function `half` is an auxiliary to `log` and it is of the class of simple recursion, and thus it is easy to prove it terminates. What is more interesting is the way to prove that `log` terminates. The function `log` is defined below:

```
half : nat -> nat
half(n) ==
  if (n = 0) then 0
  elseif (n = 1) then 0
  else 1 + half(n-2)
measure id;

log : nat -> nat
log(n) ==
  if (n = 0) then 0
  elseif (n = 1) then 0
  else 1 + log(half(n-2) + 1)
measure id;
```

The generated PO for `log` is:

```
forall n:nat &
    not(n=0) and not(n=1) => id(n) > id(half(n-2) + 1)
```

To prove the termination of `log`, the termination of `half` should be proved in advance. Being a simple function, the termination proof of `half` can be easily achieved. Having proved the termination of `half`, the following lemma, called an induction lemma, can be very useful in the termination proof of `log`.

```
forall n:nat. id(n) >= id(half(n))
```

This lemma is useful because it facilitates an almost direct proof of `log` termination. Normally, induction lemmas like this can be proven by recursive induction.

Having being proven, the induction lemma can then be added to the assumptions of the PO of `log` (using the mathematical syntax).

$$
\begin{aligned}
& \forall x : \text{nat}.\ x \neq 0 \wedge x \neq 1 \implies \text{id}(x) > \text{id}(\text{half}(x-2)+1) \\
\Leftrightarrow\ & \{\ \text{defn id}\} \\
& \forall x : \text{nat}.\ x \neq 0 \wedge x \neq 1 \implies x > \text{half}(x-2)+1 \\
\Leftrightarrow\ & \{\text{defn half backwards}, x \neq 0 \wedge x \neq 1\} \\
& \forall x : \text{nat}.\ x \neq 0 \wedge x \neq 1 \implies x > \text{half}(x) \\
\Leftrightarrow\ & \{\text{induction lemma, using defn id}\} \\
& \text{true}
\end{aligned}
$$

It is not always guaranteed that the induction lemma of this kind will help in the proof but in most cases this is convenient.

5.3 Nested Recursion over Itself

There is another type of nested recursion. When the function calls itself several times in the nested recursion. This type can be divided furthermore as functions that do not need their own semantics considered for termination require a different approach to the ones whose semantics need to be considered.

Functions whose semantics are not involved in the proof. Some nested functions do not need the nested definition to be explored in order to have their termination proved. The well known Ackerman function can elucidate this point. This is also a typical function for which termination can only be proved by using a lexicographic order.

```
id2 : nat * nat -> nat * nat
id2(m,n) == mk_(m,n);

ack : nat * nat -> nat
ack(m,n) ==
  if m = 0 then n + 1
  else if n = 0 then ack(m-1,1)
  else ack(m-1,ack(m,n-1))
measure id2;
```

For this function, three POs are generated, corresponding to the three recursive calls:

```
PO1: (forall m : nat, n : nat &
  not (m = 0) => n = 0 =>
  id2(m, n) (LEX2 >) id2(m - 1, 1))

PO2: (forall m : nat, n : nat &
  not (m = 0) => not (n = 0) =>
  id2(m, n) (LEX2 >) id2(m, n - 1))

PO3: (forall m : nat, n : nat &
  not (m = 0) => not (n = 0) =>
  id2(m, n) (LEX2 >) id2(m - 1, ack(m, n - 1)))
```

In PO2 the first parameter is unchanged and thus a lexographical order is necessary. Because of the use of the lexicographic order, PO3 becomes solvable without the second element of the tuple needing further investigation. Thus there is no need to involve the definition of ack in this proof. This is opposite to what happens in the example below.

Functions whose semantics are involved in the proof. This is the kind of function that generates proof obligations that contain a reference to the function itself. They cannot be solved by the method presented above. One example of a function of this type is nest that is defined in subsection 3.2. The problematic proof obligation is the second one:

```
forall n:nat & not(n=0) ==> id(n) > id(nest(n-1))
```

To be able to prove this proof obligation we have to assume that nest is well defined and thus terminating which is the proof we want to make in the first place. So the proof of termination of such nested functions incurs in a circular dependency between the termination proof and its definition. This kind of function termination proof is extremely difficult to automate since they normally need a property to be supplied that varies from function to function. Different ways to deal with these cases are presented in [17].

To prove the validity of the second PO, the semantics of nest has to be explored.

Since this kind the proof of this kind of function changes much from function to function, an example of proof is not included in the next section.

5.4 Mutual Recursion

As explained in Section 3.3, the generated POs fall into the same categories as shown in previous sections, the only difference is that they need to be proved for both functions in order to assure proper termination.

6 Examples of Proof

In this section, some of the examples provided during the paper are proved in HOL. The translation from VDM to HOL is made with the help of the VDM to HOL translator described in [4] and in accordance with the process shown in figure 1.

The examples show the different approach to two different kinds of proof and could be used as recipes when tackling these kinds of proof.

6.1 Straightforward Recursion

It is shown here how simple recursive functions (like factorial) can be handled. They are normally simple to prove valid or invalid. This is the class that HOL easily proves automatically too. The proofs normally consist of just making rewrites of the definition and applying arithmetic rules.

Going back to the factorial example we have the following PO that can be introduced into the goal stack as shown below. The definitions presented were obtained using the *VDM to HOL* translator directly.

Just below is shown how `id` is translated to HOL

```
- Define `id (id_parameter_1:num) = (let x = id_parameter_1 in x)`;
Definition has been stored under "id_def".
> val id_def =
    |- !id_parameter_1. id id_parameter_1
      = (let x = id_parameter_1 in x) : thm

- BasicProvers.export_rewrites(["id_def"]);
> val it = () : unit
```

This last command tells the prover to use the definition of `id` when it is doing rewrites. As seen above, in the process of translation, a bit of complexity is introduced in the function due to the translator being more general but the function is semantically equivalent to the one written in VDM.

Applying the VDM to HOL translator to the factorial PO, the following HOL formula is generated. The `g` command is used in HOL to introduce a proof in the proving mechanism in order to start the proof.

```
- g `(!uni_0_var_1
((((inv_num uni_0_var_1)  /\  (?n.(n  = uni_0_var_1 )) )  /\ T )  ==>
 (let n = uni_0_var_1 in
 ((~ (n  = 0 )) ==> ((id n)  > (id (n  - 1 )) ) )) ))`;
> ...
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !uni_0_var_1.
           (inv_num uni_0_var_1 /\ ?n. n = uni_0_var_1) /\ T ==>
           (let n = uni_0_var_1 in ~(n = 0) ==> id n > id (n - 1))
```

After introducing the formula to be proven in the goal stack, the proof is ready to be started. Again the translator introduces complexity, here *inv_num* is a predicate inserted that asserts that *uni_0_var1* is a numeric type.

The HOL command `e`, is used to execute proof steps, in this case the chosen step is to rewrite the formula using arithmetic adjustments, the definition of **`let`** and the definition of the measure (`id_def`), the latter having been added previously to be automatically used by the rewrites.

```
- e (RW_TAC arith_ss [boolTheory.LET_DEF]);
Ok..
...
   Initial goal proved.
   |- !uni_0_var_1.
        (inv_num uni_0\_var_1 /\ ?n. n = uni_0_var_1) /\ T ==>
        (let n = uni_0_var_1 in ~(n = 0) ==> id n > id (n - 1))
```

This is all that it takes to prove that factorial does indeed terminate and all these steps can and have been automated.

6.2 Nested Recursion

Recursion of this type involves proving that the argument of the call after being transformed by another function is still smaller than the input (wrt. the measure function). The example presented is the *log* function from section 5.2. The function *half* is an auxiliary to *log* and it is of the class of simple recursion thus it is easy to prove that it terminates. What is more interesting is the way to prove that *log* terminates.

The VDM proof obligation for *log* is:

```
forall n:nat & not(n=0) and not(n=1) => id(n) > id(half(n-2) + 1)
```

Which translates to HOL as the following:

Listing 1.1. PO translated using VDM to HOL translator

```
- g `(!  uni_0_var_1.((((inv_num uni_0_var_1)
  /\ (?  n.(n  = uni_0_var_1 )) )  /\ T )  ==>
  (let n = uni_0_var_1 in (((~  (n  = 0 ))  /\ (~  (n  = 1 )) )
  ==> ((id n)  > (id (1  + (half (n  - 2 )) )) ) )) ))`;
> ...
```

To prove this in HOL, first the termination of *half* should be proved. Since *half* is a function with simple recursion, the termination proof can be easily archived. Having proved the termination of *half*, the following lemma, called an induction lemma, can be very useful in the termination proof of *log*.

```
!(n:num). id(n) >= id(half(n))
```

This lemma is useful because it facilitates an almost direct proof of the termination of *log*. Normally, induction lemmas like this can be proven by induction. In HOL this can be made using the tactic *recInduct* over the induction theorem of *half*. To get the induction theorem of *half*, the function must be defined in HOL.

```
- Define `half (half_parameter_1:num)  =
  (let n = half_parameter_1 in
    (if (n  = 0 ) then 0
     else (if (n  = 1 ) then 0 else ((half (n  - 2 )) + 1 ))
     )
   )`;
> ...
```

The HOL command `Define` can only be used if the termination of the function in question can be automatically proved in HOL. Otherwise it will involve defining the function using the *Hol_defn* command and the subsequent manual proof by the user of the termination clause. It is only possible to extract the induction theorem for a function after its termination is proved.

```
- g `!(n:num). id(n) >= id(half(n))`;
> ...
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !n. id n >= id (half n)
```

Using recursive induction and rewrites it is normally possible to prove this kind of property. It requires the use of several tactics but they are all basic rewrites and simplifications, except for the induction command *recInduct*. The `THEN` tactic is an infix binary operator with two tactics as arguments (`T1` and `T2`) that applies `T1` to a goal and then `T2` to the subgoal generated.

```
- e ( rewrites... THEN
  recInduct (fetch "-" "half_ind") THEN
  rewrites ...)
> val it =
    Initial goal proved.
    |- !n. id n >= id (half n)

- val half_leq = top_thm();
> val half_leq = |- !n. id n >= id (half n) : thm
```

The first thing to do is to rewrite the measure function and simplify the 'let' expressions. Afterwards the induction lemma is applied. With a couple of additional rewrites, the goal is proved. In the end, the theorem is given a name (in this case *half_leq* and removed from the goal stack using the command *'top_thm'*.

Since the induction theorem is now proved, it can be used as an assumption when proving the termination of *log*. In HOL, an assumption can be made using the *ASSUME_TAC* command. It allows an assumption to be made, based on a theorem that has already been proved.

Now it is possible to tackle the proof obligation for *log* easily using the induction lemma. The VDM proof obligation for *log* when translated was presented before in listing 1.1. The tactics needed to prove this PO are:

```
- e ( rewrites... THEN
  ASSUME_TAC (Q.SPEC `n-2` half_leq) THEN
  rewrites ...);
>  ...
  Initial goal proved.
```

The induction lemma theorem has to be specialized (using *Q.SPEC*) to the argument that matters, in this case '*n-2*'.

```
forall n : nat & id(n) >= id(half(n)) => id(n-2) >= id(half(n-2))
```

This HOL proof follows the same steps as the proof in section 5.2.

This kind of proof can be done with higher or lower degree of automation depending on complexity of the inner function.

6.3 Mutual Recursion

As explained in section 5.4, this case does not add anything new to the picture, but just for the sake of completeness we include an example here. The POs to prove here are presented in section 3.3. Both of the POs from Section 3.3 are added to the goal stack. Prior to that, the definition of the measures function has been inserted in HOL. Here both POs are added in one go, using a conjunction, in practice that would not happen but since the tactics to apply are the same it makes the proof smaller.

```
- g `(! uni_0_var_1.((((inv_num uni_0_var_1) /\
   (? n.(n = uni_0_var_1 )) ) /\ T ) ==>
   (let n = uni_0_var_1 in ((~ (n = 0 ))
   ==> (((m_even n) > (m_odd n) )
/\
   (! uni_1_var_1.((((inv_num uni_1_var_1) /\
   (? n.(n = uni_1_var_1 )) ) /\ T ) ==>
   (let n = uni_1_var_1 in ((~ (n = 0 ))
   ==> ((m_odd n) > (m_even (n - 1 )) ) )) )) ) )) ))`;
> val it =
    Proof manager status: 1 proof.
```

Then by applying simple rewrites, both goals are proven.

```
- e (RW_TAC arith_ss [boolTheory.LET_DEF,m_even_def,m_odd_def]);
OK..
> val it =
    Initial goal proved.
```

7 Related Work

Similar work has been done before, but for VDM this is the first attempt to generate termination proof obligations for termination of recursive functions.

The VDM to HOL translator converts VDM specifications into HOL and and provides tactics to prove generated proof obligations and is defined in [4]. It can be used to prove static inconsistencies in VDM models. However termination inconsistencies are not considered in this work.

The tactics to prove termination of recursive functions used on this paper, were first suggested in [16]. Here the best way to approach the proof of termination of nested and mutually recursive algorithms is described.

PROSPER was a Proof Engine developed in HOL98 which aimed to support formal proofs for industry-standard languages, like VDM and VHDL, by making proofs invisible to the end user [18]. A translator from VDM to HOL was defined in [19] and also tactics to prove the generated proof obligations [20]. These documents can be found at the `www.vdmportal.org`. Here termination proof obligations are also ignored.

PVS (Prototype Verification System) [21] is an environment for specification and verification consisting of a specification language, a parser, a type checker and an

interactive theorem prover. The PVS type checker generates TCCs (Type Correctness Conditions[1]) that contemplate termination of recursive functions. However PVS allows less expressiveness because the modeling of mutual recursive functions is not allowed [22].

8 Concluding Remarks and Further Work

This work has enabled the automatic generation of proof obligations ensuring the termination of recursive functions in VDM models and these have also been moved over to HOL enabling verification of them. In the process a new **measure** keyword was introduced in VDM. In addition it has categorised the different kinds of recursive definitions with respect to the ability to automate their verification of their termination inside HOL. The novelty is primarily in the treatment of mutual recursion between functions which has been introduced. For example in PVS it is not at all possible to enable such mutual recursion.

VDM also contains polymorphic functions and the recursion termination proof obligations for these has also been completed. However, Curried functions have not yet been taken into account so that is a part of the future work that is currently being undertaken. Finally from an end-user perspective it would be convenient if it was possible to express the status of proofs carried out in HOL at the VDM level that the user is familiar with. This is not yet done so this also belongs in the future work category.

The features of the proof obligation generation for the termination of recursive functions are available both inside VDMTools as well as in Overture.

Acknowledgments. We would like to thank Nick Battle, Luis Barbosa, Miguel Ferreira and Hugo Macedo and the anonymous referees for feedback on the work reported in this paper.

References

1. Aichernig, B.K., Larsen, P.G.: A Proof Obligation Generator for VDM-SL. In: Fitzgerald, J.S., Jones, C.B., Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 3–540. Springer, Heidelberg (1997) ISBN 3-540-63533-5
2. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
3. Ribeiro, A.: An Extended Proof Obligation Generator for VDM++/OML. Master's thesis, Minho University with exchange to Engineering College of Arhus (July 2008)
4. Vermolen, S.: Automatically Discharging VDM Proof Obligations using HOL. Master's thesis, Radboud University Nijmegen, Computer Science Department (August 2007)
5. Vermolen, S., Hooman, J., Larsen, P.G.: Automating Consistency Proofs of VDM++ Models using HOL. In: Proceedings of the 25th Symposium On Applied Computing (SAC 2010), Sierre, Switzerland. ACM Press, New York (March 2010)
6. Gordon, M.: HOL: A Proof Generating System for Higher-Order Logic. In: Birtwistle, G., Subrahmanyam, P.A. (eds.) VLSI Specification, Verification, and Synthesis, Kluwer Academic Publishers, Dordrecht (1987)

[1] This is just a different name for a Proof Obligation.

7. Boyer, R.S., Moore, J.S.: A Computational Logic. Academic Press, London (1979)
8. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object–oriented Systems. Springer, New York (2005)
9. Fitzgerald, J., Larsen, P.G.: Modelling Systems – Practical Tools and Techniques in Software Development. Cambridge University Press, Cambridge (1998) ISBN 0-521-62348-0
10. Barringer, H., Cheng, J.H., Jones, C.B.: A Logic Covering Undefinedness in Program Proofs. Acta Informatica 21, 251–269 (1984)
11. McCarthy, J.: A Basis for a Mathematical Theory of Computation. In: Braffort, P., Hirstberg, D. (eds.) Western Joint Computer Conference. Then published in: Computer Programming and Formal Systems, pp. 33–70. North Holland, Amsterdam (1961)
12. Elmstrøm, R., Larsen, P.G., Lassen, P.B.: The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications. ACM Sigplan Notices 29(9), 77–80 (1994)
13. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in Support for Formal Modeling in VDM. Sigplan Notices 43(2), 3–11 (2008)
14. Larsen, P.G., Battle, N., Ferreira, M., Fitzgerald, J., Lausdahl, K., Verhoef, M.: The Overture Initiative – Integrating Tools for VDM. ACM Software Engineering Notes 35(1) (January 2010)
15. Aichernig, B.: A Proof Obligation Generator for the IFAD VDM-SL Toolbox. Master's thesis, Technical University Graz, Austria (March 1997)
16. Giesl, J.: Termination of nested and mutually recursive algorithms. Journal of Automated Reasoning 19, 10–29 (1997)
17. Slind, K.: Another look at nested recursion. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 498–518. Springer, Heidelberg (2000)
18. Dennis, L.A., Collins, G., Norrish, M., Boulton, R., Slind, K., Robinson, G., Gordon, M., Melham, T.: The PROSPER Toolkit. In: Schwartzbach, M.I., Graf, S. (eds.) TACAS 2000. LNCS, vol. 1785, Springer, Heidelberg (2000)
19. Agerholm, S., Sunesen, K.: Formalizing a Subset of VDM-SL in HOL. Technical report, IFAD (April 1999)
20. Agerholm, S., Sunesen, K.: Reasoning about VDM-SL Proof Obligations in HOL. Technical report, IFAD (1999)
21. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS System Guide - Version 2.4 (2001)
22. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference - Version 2.4 (2001)

Correct-by-Construction Model Transformations from Partially Ordered Specifications in Coq*

Iman Poernomo and Jeffrey Terrell

Department of Computer Science, King's College London,
Strand, London WC2R 2LS, UK
iman.poernomo@kcl.ac.uk, jeffrey.terrell@kcl.ac.uk

Abstract. This paper sketches an approach to the synthesis of provably correct model transformations within the Coq theorem prover, an implementation of Coquand and Huet's Calculus of Inductive Constructions. It extends work done by Poernomo on *proofs-as-model-transformations* in the related formalism of Martin-Löf predicative Constructive Type Theory. We show how the impredicative theory of Coq, together with its treatment of coinductive types, lends itself to the synthesis of a wider range of model transformations than Poernomo had treated before. We illustrate the practical benefits and potential scalability of our approach by means of a case study taken from industry.

1 Introduction

This paper presents an approach to the synthesis of provably correct model transformations within the Coq theorem prover, an implementation of Coquand and Huet's Calculus of Inductive Constructions.

Model transformations within the Model Driven Architecture (MDA) paradigm are meant to enable designers to focus most of their time on providing a robust, architecturally sound platform independent model. Then, given a particular platform and associated metamodel choice, a designer applies a (possibly off-the-shelf) transformation to automatically obtain a specific, refined model. In this way, MDA eliminates some of the effort spent on making implementation decisions at the specification level, and, ideally, results in a better generated platform specific model than that obtained from a manual process.

MDA is increasingly being taken up by industry and, as a consequence, transformations of interest are becoming more complex and mission critical. It becomes essential, therefore, to have maximal levels of trust in the correctness of model transformations. The informality of MDA as it currently stands makes the approach untrustworthy and dangerous. If model transformations are incorrect, the MDA process can result in software of a lower quality than that produced by traditional software development. A small number of errors in the composition of

* This work was supported by Kennedy Carter Ltd and the Engineering and Physical Sciences Research Council (EPSRC grant EP/G03012X/1, "Higher-Order Refinement Techniques for Model Driven Architecture").

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 56–73, 2010.
© Springer-Verlag Berlin Heidelberg 2010

a number of complex transformations can easily lead to an exponential number of errors in the code, that are difficult to trace and debug.

We solve this problem by leveraging a property of Constructive Type Theory known as the Curry-Howard Isomorphism, where data, functions and their correctness proofs are treated as ontologically equivalent, and where a similar equivalence holds for the related trinity of typing information, program specifications and programs. A practical implication of the isomorphism is that, by proving the logical validity of a model transformation specification, we can automatically synthesize a model transformation that satisfies the specification. Following [7], we call this implication the *proofs-as-model-transformations* paradigm.

This paper presents a significant advance over our previous work in Martin-Löf type theory. Our previous work considered simple specifications of model transformations as relationships between source and target metamodel instances, but gave no detail on how a complex transformation might be derived if its specification involves *multiple* mappings between metamodels built from a large number of interrelated metaclasses. We now give a treatment of a subset of such specifications, which we call *partially ordered specifications*. In such specifications, source and target models can take on any graph structure, with metamodels allowing, as usual, bidirectionality and circular references. However, the *transformation specification* itself is given as a series of mappings, defined over each metaclass of the model via a partially ordered traversal of the source metamodel graph, from an given initial metaclass, with the target model requirements also provided according to a partially ordered traversal of the target metamodel graph. We contend that partially ordered specifications are sufficient to define the majority of industrially useful transformations.

We illustrate the practical benefits and potential scalability of our approach by presenting the end results of a case study we have conducted, involving part of a transformation used at Kennedy Carter for safety critical system development (the domain specific properties of the case study have been obfuscated for confidentiality reasons).

This paper assumes the reader is familiar with the UML representation of classes, relationships and objects and has a partial familiarity with the MOF specification document [5]. A detailed study of constructive type theory can be found in [2] or [8] (we follow the formulation of the latter here). Section 2 presents a brief overview of how proofs and programs are related within the type theory of Coq. Section 3 describes our approach to metamodelling and model transformation development and Section 4 provides an example. Finally, Section 5 contains our conclusions.

2 Proofs and Programs in Coq

This section presents a brief summary of the Calculus of Inductive Constructions (CIC), the formalism implemented by Coq.

2.1 Values and Types

The CIC is based on the principle that everything is a value of a particular type, where values include constants and functions, and types range over

- ordinary inductive data types, such as the booleans `bool` and the natural numbers `nat`;
- co-inductive types, such as the list of natural numbers starting from `10` (an example of an infinite object);
- parametrized types, such as the list of natural numbers `1::2::3::nil`;
- higher order types, such as the type of all basic types `Set`, and the type of all logical propositions `Prop`.

If `v` is a value and `t` is its type, we write `v: t`. For example:

- the natural number `3` is written `3: nat`[1];
- the addition function `plus`, which takes two natural numbers and returns a natural number, is written `plus: nat -> nat -> nat`;
- the logical statement "there exists a natural number greater than zero" is written `exists y: nat, ge y O: Prop`;
- the logical predicate `ge`, which takes two natural numbers and returns a `Prop`, is written `ge: nat -> nat -> Prop`.

2.2 Internal Programming Language (the Lambda Calculus)

Coq's *values* range over a theory of predefined constants and functions (such as the natural numbers and arithmetic functions) and a *lambda calculus* for building new functions. In this calculus, which is essentially an internal functional programming language with a syntax similar to Haskell or SML, functions can be constructed from built-in APIs, and various constructs exist for dealing with recursion, disjoint unions, matching by cases, record types and so on.

An important aspect of this language, as with all functional programming languages, is its support for anonymous functions. In conventional mathematical notation, the lambda abstraction $\lambda x\colon nat.x + x$ defines a function that inputs $x\colon nat$ and returns $x+x$. In Coq, this function is written `fun x: nat => x + x` and its type is `nat -> nat` (the type of functions that take in naturals as input and return naturals as output). Furthermore, since functions can be applied to inputs, the application of `fun x: nat => x + x` to `3: nat`, say, which in its unevaluated form is written `(fun x: nat => x + x) 3`, is clearly a value of type `nat`.

Evaluation is formalized by rules that define how terms can be converted or *reduced*, into simpler terms. The transitive closure of these reduction rules define the system's *normalization* relation, providing the operational semantics for computing the final value of any function application. A key principle of

[1] The symbol `3` is an abbreviation for the expression `S(S(SO))`, where `S` is the successor function, and `O` (the letter, that is) is a representation of zero.

Coq is *strong normalization*: that *all* possible sequences of applications of reduction rules are confluent and terminate. We do not admit terms that could lead to infinite reduction sequences (for example, infinite recursion). As a result, normalization provides a *canonical, normal value* for each term. The lambda calculus of Coq thus forms an internal functional programming language in which every program terminates. We may use it to define new programs from known programs.

2.3 Internal Logic

The CIC works with a collection of logical formulae, each of type `Prop`, which are built from a closure over basic propositions, predicates over terms, implication `A -> B`, conjunction `A /\ B`, disjunction `A \/ B`, typed universal quantification `forall x: T, A` and existential quantification `exists x: T, A`. The symbols `->`, `/\` and `\/` correspond to the usual logical connectives, and the keywords `forall` and `exists` correspond to the quantifiers $\forall$ and $\exists$.

The connectives and quantifiers allow us to define well-typed logical propositions within the calculus, and because propositions may involve predicates over terms, we can define propositions that make statements about our lambda calculus (that is, we can specify properties of programs). But, in and of themselves, the logical formulae do not allow us to *reason*: we need rules to define a proof system in order to do that. We will return to this question shortly.

2.4 Co-inductive Types

Co-inductive types merit special attention because they play an important part in the representation of metamodels. A co-inductive type, just like any other type, is defined by "prescribing what we have to do to construct an object of that type" [4]. Let us start by defining a particular co-inductive type, namely

```
CoInductive X : Set :=
   Build'X : nat -> Y -> X
with Y : Set :=
   Build'Y : nat -> X -> Y.
```

This type definition (which actually defines the type of a pair of *mutually* co-inductive types `X` and `Y`, because `X` refers to `Y` and `Y` refers to `X`) tells us that there is only one way in which an object of type `X` can be constructed, and that is to call `Build'X` with an object of type `nat`, and an object of type `Y`. It also tells us something similar about `Y`.

Consider a specific object `x1` of type `X` (as depicted in Fig. 1), which is constructed by evaluating `f 1`, where `f` is defined by

```
CoFixpoint f (n : nat) : X :=
   Build'X n (Build'Y (S n) (f (S (S n)))).
```

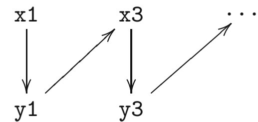


Fig. 1. Object x1 and its followers

Clearly, f is recursive (it appears on both sides of the definition) and x1 is infinite because the evaluation of f 1, of which the first few terms are

```
Build'X 1 (Build'Y 2 (Build'X 3 (Build'Y 4 ...,
```

never terminates. However, this does not prevent us from reasoning about x1, so long as we restrict ourselves to finite parts or arbitrary elements of x1. In that way, our computations are bound to terminate. For example, let us find the object of type nat that was used to construct the object of type X that follows a specified object of type X. Two navigation functions and an access function are required, namely

```
Definition nav'x (y : Y) : X :=
   let (n, x) := y in x.

Definition nav'y (x : X) : Y :=
   let (n, y) := x in y.

Definition val'x (x : X) : nat :=
   let (n, y) := x in n.
```

nav'x navigates from an object of type Y to the following object of type X, and nav'y does something similar. Furthermore, val'x picks off the object of type nat that was used to construct a specified object of type X. Applying a suitable composition of functions to x1 yields 3, the nat that was used to construct x3, the object of type X that follows x1.

```
Eval compute in (val'x (nav'x (nav'y (f 1)))).
= 3 : nat
```

Let us now consider a slightly different example, which is more germane to the representation of metamodels, by constructing a mutually referential pair of co-inductive objects x1 and y2 (see Fig. 2) as follows:

```
CoFixpoint x1 : X :=
   Build'X 1 y1
with y1 : Y :=
   Build'Y 2 x1.
```

Starting from x1, it is clearly possible to navigate to y2 and back again, an arbitrary number of times. As in the previous example, therefore, x1 is an infinite

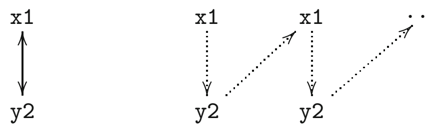


Fig. 2. Mutually referential objects `x1` and `y2`, and navigations back and forth

object. Performing a dual navigation from `x1` followed by an access yields, as expected, `1`.

```
Eval compute in (val'x (nav'x (nav'y x1))).
= 1 : nat
```

What this example shows is that co-inductive types can be used to represent mutually referential objects, of the kind that inhabit most metamodels. We shall discuss this in more detail in Section 3.1.

2.5 Higher-Order Values

Coq implements a higher-order type theory in the sense that it contains *higher-order values* that can be used to *type* other values.

Propositions also possess this dual status. Not only are they values of type `Prop`, but they also stand as *types* of values. The idea is that the inhabiting values of a proposition are the possible *proofs* of that proposition. If a proposition has at least one inhabiting value, then it is said to be true (because it has a proof). In this way, the intended meaning of the various connectives is *constructive*. That is, we understand a formula F to be *true* only if we can construct a proof p that supports or realises its truth, written $p : F$. Rules of inference are included in a straightforward way to accommodate theories of various data types, as is reasoning about the simply typed lambda calculus, induction and recursion. We do not provide these rules here – the reader is referred to [8] for details.

2.6 Extracting Programs from Proofs

The Curry-Howard isomorphism supports the notion of proof-carrying code.

Theorem 1 (Program Extraction). Let `forall x: T, exists y: U, P(x, y)` be a well-formed formula built from well-defined predicates and functions in Coq's type theory. There is a mapping `extract` from proof-terms to *simply typed lambda terms* (terms that do not involve logical propositions) such that, if

$$\vdash \texttt{p: forall x: T, exists y: U, P(x, y)}$$

is a well typed term, then

$$\vdash \texttt{forall x: T, P(x, extract(p) x)}$$

is provable.

Space does not permit us to describe the extraction mapping, but essentially it is developed using the generic machinery of [8]. The implication of this theorem is that, given a proof of a formula `forall x: T, exists y: U, P(x,y)`, we can automatically synthesize a lambda calculus *program* `f` that, given input `x: T`, will *correctly* produce an output `f x` that satisfies the constraint `P(x, f x)`.

3 Doing MDA in Coq

Our notion of proofs-as-model-transformations essentially follows from Theorem 1. A model transformation can be specified as a constraint in the OCL, over instances of an input PIM and an output PSM written in the Meta-Object Facility (MOF) [5], or any comparable constraint and class-based metamodelling languages. Assuming that we can develop types to represent the PIM and PSM metamodels, and that we can write the constraint as a logical formula over the metamodels, we can then specify the transformation as a `forall exists` formula. Then, in order to synthesize a provably correct model transformation, we can prove the formula's truth and apply the extraction mapping according to Theorem 1.

The main technical challenges posed by this approach are twofold. First, the formalization of MOF-based metamodels as types (as input/output types of transformations) is not clear. Second, a complex model transformation typically has a complex specification that will not have a straightforward representation or proof in Coq. Are there common patterns of specification definition and proof (and, consequently, synthesis) that can assist us in dealing with this complexity? These two challenges are now addressed.

3.1 Encoding Metamodels as Types

In [6], metamodel/model instantiation relationships of the MOF were treated using terms and types within the predicative type hierarchy of a Martin-Löf Type Theory. That work suggested a particular co-inductive encoding for bidirectional metaclasses, which required specialized rules to be added to the type theory. Here, we employ a variant implementation, which makes use of the co-inductive type definition schemas built into Coq. The advantage of this new approach is that we readily have an implementation of the metamodels-as-types idea within a robust tool kit.

Our approach is to define a Coq type for each metamodel ModelLang, so that $\vdash$ *model* : ModelLang is derivable if, and only if, the term *model* corresponds to a well formed model instance of the metamodel. Model transformations can then be defined as lambda functions within the internal programming language of Coq, typed by metamodel types.

We employ co-inductive types to represent metaclass references for both unidirectional and bidirectional associations.[2] This makes it possible to define the

[2] In very special circumstances, inductive data types can also be used.

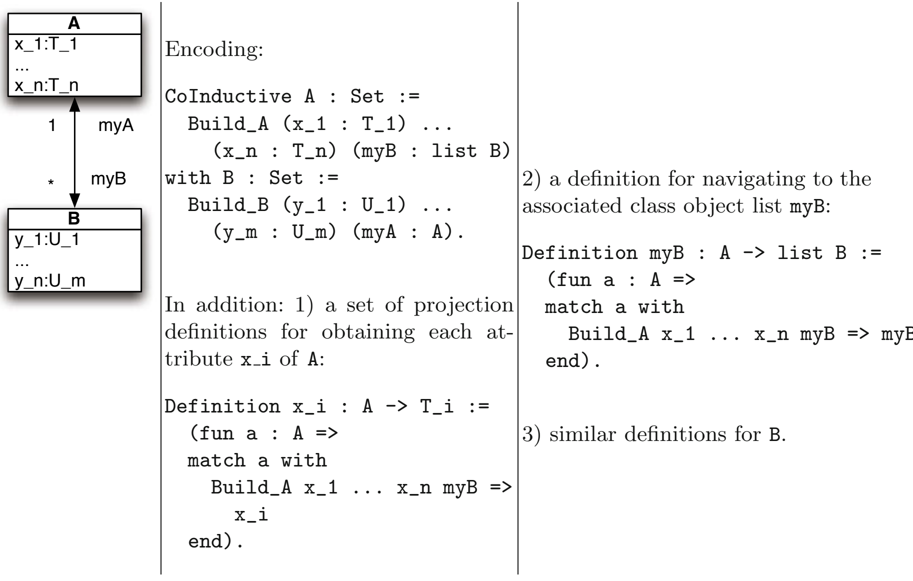


Fig. 3. Co-inductive encoding of metaclass structures involving a bidirectional reference

graph of any metaclass diagram using the `CoFixpoint` operator. For example, the two metaclasses that participate in the one-many bidirectional association in Fig. 3, are encoded as co-inductive types, which utilise the inductive parametrized `list` type to represent multiplicities. The fact that metaclass `A` has a reference to many objects of metaclass `B`, is represented by the constructor of metaclass `A` having a list attribute `myB` of type `list B`.

With such an encoding, it is a simple matter to instantiate a model with (say) an instance `a_1` of metaclass `A` linked in both directions to instances `b_1` and `b_2` of metaclass `B`, as follows:

```
CoFixpoint a_1 : A :=
   Build_A v_1 ... v_n b_1::b_2::nil
with b_1 : B :=
   Build_B q_1 ... q_m a_1
with b_2 : B :=
   Build_B r_1 ... r_m a_1.
```

The encoding of conditional one-one relationships and generalizations, which we have omitted in this paper, can be accomplished using Coq's `option` and `sum` types. Please follow the link in Section 5 for further details.

3.2 Constraints

In general, a MOF metamodel consists of a metaclass structure over which a set of OCL constraints (defining well-formedness and semantics) range. Fortunately,

there is a straightforward mapping from OCL constraints over metamodel elements into logical propositions of type `Prop` in Coq. Utilizing this mapping, it is possible to form a new *subset type* that represents the complete metamodel as a higher-order type, by pairing the structural information given by the encoding of a metamodel `MM` (say) with its constraints `B` (say), as a term `{x: MM | B}` of type `Set`, where `x: MM` is a typed variable, and `B: MM -> Prop` is a predicate function over `MM`.

The general reader may surmise the type's intuitive meaning from ordinary mathematical set theory, since any term `aModel` of type `{x: MM | B}` must not only instantiate the metaclass structure of type `MM`, but it must also satisfy `B[aModel/x]`. However, the difference in Coq is that inhabitation of this type requires *constructive evidence*: that is, an element `aModel` of type `{x: MM | B}` is necessarily a term of the form

```
exist (fun x: MM => B) w p,
```

where `exist` is a constructor that pairs together two important elements: a witness `w` and a proof `p`, which provides the evidence that `B[w/x]` is true. In other words, any instantiation of a metamodel type must include a proof that certifies its conformance to the metamodel's constraints. In this sense, our formalism promotes certified metamodelling.

3.3 Specifications

Whether we intend to handcraft the implementation of a transformation or formally synthesize it, first we need to specify what we expect of the transformation. As we have shown in [7], a simple model transformation between instances of metamodels `M` and `N`, can be specified as a `forall exists` formula of the form

```
forall x: M, Pre(x) -> exists y: N, Post(x, y),          (1)
```

where `Pre(x)` specifies an assumed precondition over instance `x` of metamodel `M`, and `Post(x, y)` prescribes the required input-output relationship between instances `x` and `y` of metamodels `M` and `N` respectively. By Theorem 1, if we can derive a proof of (1), then a provably correct transformation that satisfies the pre- and post-conditions can be obtained.

In [7] we demonstrated that this approach is applicable to the development of transformations from contractual specifications: given a contractual specification written in a first-order language such as the OCL over the input and output metamodel vocabularies, we can map it to a formula of the form (1). That treatment is what might be called an *unstructured*, *single* arbitrary form of contractual specification: one precondition and one postcondition, each potentially very large, predicating over the entirety of two metamodels.

The main problem with that approach is that, in practice, a model transformation is *not* specified as a single contract, but as a *sequence* of interrelated contracts between the elements of the source and target metamodels. When using a declarative model transformation language, if the transformation is sufficiently

simple, these specifications can then be taken to be the actual transformation itself. Our approach is not used for synthesis of declarative transformations, but of algorithmic transformations. However, even in the case of algorithmic transformation languages such as Kermeta or the Executable UML toolkit of Kennedy Carter, a specification is always divided and conquered via a modular hierarchy of contractual specifications.

Further, from the perspective of the proofs-as-model-transformations paradigm, a disadvantageous side-effect of beginning with an unstructured, single arbitrary contractual specification, is that while we may have a straightforward mapping to the form of (1), we unfortunately have no heuristics to guide us in deriving its proof and, consequently, in extracting the correct transformation.

By focusing on how a transformation is typically given, on the common interdependencies of the hierarchy of modular requirements, we can also discover common patterns of proof implied by the corresponding structure of the specification in Coq.

3.4 Partially Ordered Specifications

We consider a particular form of complex transformation specification that we refer to as *partially ordered* (PO) in this paper. For the sake of illustration, we shall restrict ourselves to metamodels in which

- associations are one-many;
- metaclasses are associated with at least one other metaclass;
- there is at most one association between any two classes, but their ends may be of any multiplicity;
- there are no generalizations.

However, there is nothing significant about these restrictions.

Definition 1 (Partial Order). Consider a MOF-based metamodel consisting of a set of metaclasses $\mathsf{Class} = \{C_1, \ldots, C_n\}$ and a set of associations $\mathsf{Assoc} = \{R_1, \ldots, R_m\}$ holding between metaclasses. Let $(C_i \; \widehat{R_k} \; C_j)$ hold if there is an association $R_k \in \mathsf{Assoc}$ between C_i and C_j. A *partial order* over the metamodel $\{H, \leq\}$ consists of

- a reflexive, antisymmetric and transitive relation $\leq$, defined over all metaclasses in C so that $C_i \leq C_j$ if, and only if,

$$C_i \; \widehat{R^1} \; C^1 \; \widehat{R^2} \; \ldots \; \widehat{R^l} \; C^l \; \widehat{R'} \; C_j$$

  for some (possibly empty) chain of associated classes $C^1, \ldots, C^l \in \mathsf{Class}$ and associations $R^1, \ldots, R^l, R' \in \mathsf{Assoc}$;
- a root metaclass $H \in \mathsf{Class}$, such that $H \leq C$ for all $C \in \mathsf{Class}$.

If $C_i \leq C_j$ and there is an association $R \in \mathsf{Assoc}$ such that $(C_i \; \widehat{R} \; C_j)$ holds, then we say that R is a *PO association* (an association that defines the partial order). If an association is not a PO association, we call it an *ancillary association* (an association that cannot be included in a navigation of the metamodel graph conducted only with respect to PO associations).

The basic idea of a PO specification is illustrated in Fig. 4, which shows a specification as a series of sub-specifications that define required mappings between individual metaclasses in a rooted directed graph. The series of sub-specifications is defined according to the partial order of the input metamodel and preserves the partial order of the output metamodel, in the sense that a subspecification over a metaclass A can only involve references to metaclasses that preceed A in the PO, and cannot involve references to metaclasses that are greater than A in the PO.

Definition 2 (PO Specification). Let Source be a metamodel consisting of a set of metaclasses $\mathsf{Source} = \{A_1, \ldots, A_w\}$ and a partial order $\{A_1, \leq_{\mathsf{Source}}\}$ constructed from its set of associations. Let Target be a metamodel consisting of a set of metaclasses $\mathsf{Target} = \{B_1, \ldots, B_v\}$ and partial order $\{B_1, \leq_{\mathsf{Target}}\}$ constructed similarly. A *PO specification* describes a model transformation ϕ : $\mathsf{Source} \rightarrow \mathsf{Target}$ via a series of OCL pre and postcondition pairs[3], that is

$$\mathsf{SPEC} = \{(Pre_i(x_i), Post_i(x_i, y_i)) \mid i = 1, \ldots, w\} \ .$$

Each pair specifies how ϕ should operate on an assumed instance x_i of A_i from Source, to produce a corresponding instance y_i of $\phi(A_i)$ from Target.

- The precondition $Pre_i(x_i)$ is a constraint that can predicate over any attribute of x_i and any aspect of $(R\ x_i)$, where R can be either a PO or ancillary association.
- The postcondition $Post_i(x_i, y_i)$, on the other hand, is restricted to predicate over any attribute of x_i or y_i and *only* navigations to, and variable instantiations of, "preceding" associated metaclasses in the POs – that is, it may predicate only over instance variables or navigated references $x' : X'$ of metaclasses $X' \leq_{\mathsf{Source}} A_i$ and instance variables or navigated references $y' : Y'$ of metaclasses $Y' \leq_{\mathsf{Target}} \phi(B)$.

The PO specification demands that ϕ is a surjective homomorphism from A to B, preserving the partial orders, so that if $A_i \leq_{\mathsf{Source}} A_j$, then $\phi(A_i) \leq_{\mathsf{Target}} \phi(A_j)$.

The transformation ϕ is then specified by SPEC to be a function with the following procedural semantics. ϕ is $\mathsf{Proc}(a_1, 1)$, where $\mathsf{Proc}(e, i)$ is defined recursively as follows.

1. Given an instance e of A_i, we add $\phi(e) : \phi(A_i)$ to the output metamodel instance, preserving $Pre_i(e)$ and $Post_i(e, \phi(e))$ in the sense defined above.
2. If $A_i \leq_{\mathsf{Source}} A_j$ and $A_i\ \widehat{R}\ A_j$ (so that R is a PO association) then, for each instance $f : A_j$ contained in the associated reference list $(R\ e)$, we repeatedly invoke $Proc(f, j)$.

Remark 1. Our definition of a PO specification is essentially generic with respect to first-order constraint language, metamodelling notation and precise definition

[3] We allow free variables in our OCL constraints.

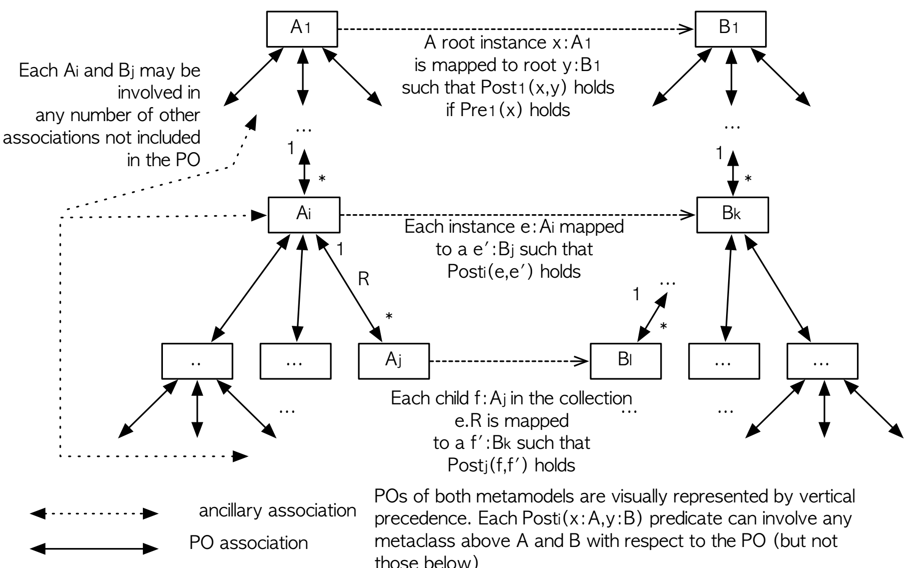


Fig. 4. Relational form of a PO specification. In general, m need not equal n and each i need not equal k for each mapping from A_ni to B_mk.

of procedural semantics. It should be clear how similar specifications could be provided using constraints written in, for example, OCL over MOF metamodels, contracts over Kermeta metamodels, and even assertions over Java representations of metamodels.

Remark 2. We have shown in [6] that OCL constraints over a MOF-based metamodel have an obvious mapping into a higher-order type theory. It is a trivial extension of that work to show that we have a mapping from such OCL constraints to formulae of type `Pop` in Coq, over elements of the metamodel's encoding. We denote this mapping in Definition 3, by changing the font of the encoded constraint to `typewriter`, and changing the format of logical connectives as appropriate.

For the purposes of developing a guaranteed correct transformation that implements the specification, we need to represent its intended meaning as a formula in Coq's type theory. This is now defined.

Definition 3 (Interpretation of PO Specification in Coq). Consider PIM and PSM metamodels Source and Target consisting of a set of metaclasses $\mathsf{Source} = \{A_1, \ldots, A_w\}$ and $\mathsf{Target} = \{B_1, \ldots, B_v\}$, with associated partial orders $\{A_1, \leq_{\mathsf{Source}}\}$ and $\{B_1, \leq_{\mathsf{Target}}\}$ derived over their associations. Let

$$\mathsf{SPEC} = \{(Pre_i(x_i), Post_i(x_i, y_i)) \mid i = 1, \ldots, w\}$$

be a *PO specification* describing a model transformation $\phi : \mathsf{Source} \rightarrow \mathsf{Target}$. Furthermore, let A_i be a metaclass from Source. We define

$$\mathtt{Traverse}(\mathtt{x} : \mathtt{A_i}) \equiv \bigwedge_{\mathtt{j}=\mathtt{f_i(1)}}^{\mathtt{j}=\mathtt{f_i(k)}} \mathtt{Traverse_j}(\mathtt{x} : \mathtt{A_i})$$

$$\mathtt{Traverse_j}(\mathtt{x} : \mathtt{A_i}) \equiv \mathtt{forall}\ \mathtt{x_j} : \mathtt{A_j},\ \mathtt{x_j} \in (\mathtt{R_j}\ \mathtt{x})\ \texttt{->}\ \mathtt{Pre_j}(\mathtt{x_j})\ \texttt{->}$$
$$\mathtt{exists}\ \mathtt{y_j} : \phi(\mathtt{A_j}),\ \mathtt{Post_j}(\mathtt{x_j},\ \mathtt{y_j})\ \texttt{/\textbackslash}\ \mathtt{Traverse}(\mathtt{x_j})$$

if there is a nonempty set $\{A_{f_i(1)}, \ldots, A_{f_i(k)}\}$ of all metaclasses, constructed so that for each $l = 1, \ldots, k$, there is a $R_{f_i(l)}$ where $(A_i\ \widehat{R_{f_i(1)}}\ A_{f_i(l)})$ and $A_i \leq_{\mathsf{Source}} A_{f_i(l)}$, with f_i forming an injection $\{1, \ldots, k\} \mapsto \{1, \ldots, w\}$. If there is no such set, then

$$\mathtt{Traverse}(\mathtt{x} : \mathtt{A_i}) \equiv \mathtt{True}\ .$$

With these definitions, the *type theoretic interpretation of the PO specification* in Coq becomes

$$\mathtt{forall}\ \mathtt{x_1} : \mathtt{A_1},\ \mathtt{Pre_1}(\mathtt{x_1})\ \texttt{->}\ \mathtt{exists}\ \mathtt{y_1} : \mathtt{B_1},\ \mathtt{Post_1}(\mathtt{x_1}, \mathtt{y_1})\ \texttt{->}\ \mathtt{Traverse}(\mathtt{x_1})\ . \tag{2}$$

We then have the following theorem.

Theorem 2 (Transformations from PO Specifications). Consider PIM and PSM metamodels Source and Target with a PO specification SPEC as assumed in Definition 3. Let $P(\mathsf{SPEC})$ be the type theoretic interpretation of the PO specification (2). If $\vdash p : P(\mathsf{SPEC})$ is a well-typed term, then there is mapping extract such that $\mathsf{extract}(p) : \mathsf{Source} \rightarrow \mathsf{Target}$ is a model transformation function that satisfies the procedural semantics prescribed by SPEC according to Definition 2.

Proof. The proof follows by an extension of that given in [8] and reasoning over the definition of the procedural semantics of PO specifications (sketched in Definition 2).

This theorem allows us to synthesize provably correct model transformations from PO specifications. Assuming that PO specifications are scalable to a wider range of model transformation requirements, then this theorem suggests that the proofs-as-model-transformations paradigm is similarly scalable.

A further implication of this theorem is that the logical structure of (2) immediately simplifies the task of proof derivation (and, consequently, of synthesis), because each conjunct of $\mathsf{Traverse}(\mathtt{x} : \mathtt{A_i})$ begins with a universal quantification over a variable $\mathtt{x_{f_i(1)}} : \mathtt{A_{f_i(1)}}$ that both inhabits a *list* of instances $(\mathtt{R_{f_i(1)}}\ \mathtt{x}) : \mathtt{list}\ \mathtt{A_{f_i(1)}}$ and satisfies the precondition. This suggests that each conjunct might most easily be derived through application of list induction. The choice of induction is generally one that a human prover must make in Coq. However, once this is done, remaining proof steps can often be automated. We have found that a typical (2) specification can then be derived by selecting induction

rules systematically as suggested by the quantification clauses, in conjunction with automated intermediate proof steps.

Remark 3. Co-induction presents the complications of a bisimular notion of equality and of infinite co-inductive schema for reasoning over the structure of metaclass types. In the case of PO specifications and the approach we take to deriving proofs from them, these complications do not present a problem for our purpose of extracting model transformations from proofs of specifications involving formal metamodel types, because the proofs are generally not made over the structure of the metamodel types. Co-induction is only important for potentially complex bidirectional navigations between elements of metaclass types, within the specification and constructive content of the proof.

4 Case Study

The majority of large-scale model transformations, covering a wide range of industrial requirements, can be defined as PO specifications. This assertion is based on the experience of the authors in model transformation development but, of course, requires a larger empirical study to verify.

In lieu of such a study, we consider an important fragment of a model-to-text transformation developed at Kennedy Carter, which distributes platform-independent UML models across multiple processes.

The input metamodel Source (the left hand side metamodel of Fig.5) is a fragment of executable UML, in which classes have operations, operations have statements and statements are either invocation statements or (simplifying for the purposes of this illustration) statements of other unspecified kinds. Each class is associated with an operating system process during system configuration, so that its operational code will run in the context of a specific process.

The purpose of the transformation is to instantiate an output metamodel Target (the right hand side metamodel of Fig.5), in which there is a package for each process, and within each package, appropriate stubs and skeleton code for each remote operation invocation.

Remote operation invocation is understood in the following sense. Let i be an invocation statement in Source and let p_i be the process that executes the class that has the operation that contains i. Furthermore, let p_j be the process that executes the class that has the operation that i invokes. Clearly, $p_i = i.R4.R3.R2.R1$ and $p_j = i.R5.R2.R1$. If $p_i \neq p_j$, the invocation is said to be remote. A remote invocation is implemented by a *stub* in the source process and a *skeleton* in the destination process, to manage the flow of control and data between the invoking and invoked operations.

Consider a PO $\{Process, \leq_{\mathsf{Source}}\}$ over the source metamodel that preserves the vertical order of Fig. 5, so that $A \leq_{\mathsf{Source}} B$ if class A is higher than B in the figure, and in which all associations are PO apart $R5$, which is ancillary.

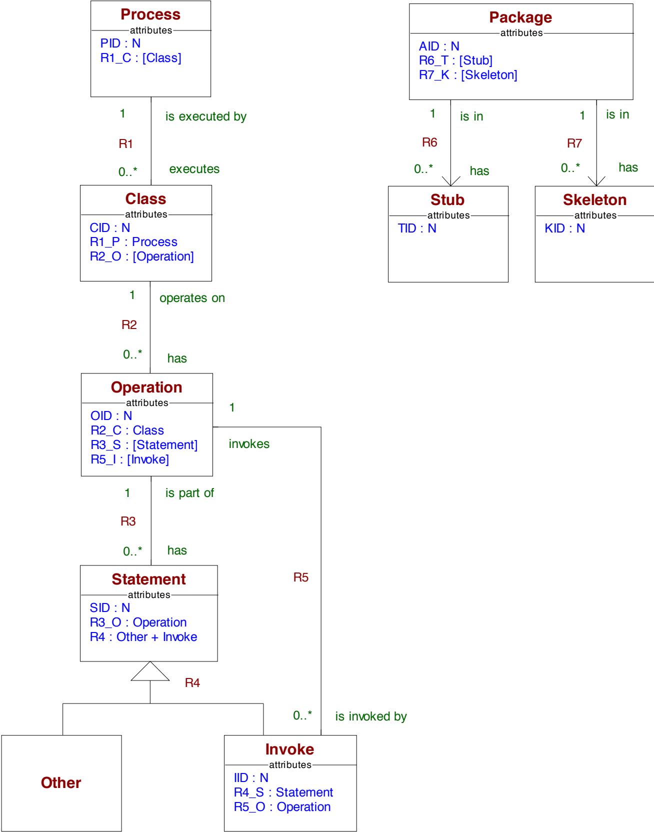


Fig. 5. The Source (LHS) and Target (RHS) metamodels

Similarly, consider a PO $\{Package, \leq_{\mathsf{Target}}\}$ over the target metamodel, in which all associations are PO.

The transformation ϕ : Source $\rightarrow$ Target is then defined by a PO specification, whose content can be written (in informal English, but a first-order logical representation is straightforward) as the following sequence of pre and postconditions over metaclasses *Process* and *Invoke* (the other pre and postconditions are nugatory and can be assumed to be "True"):

i	Source$_i$	ϕ(Source$_i$)	$Pre_i(x)$	$Post_i(x, y)$
1	*Process*	*Package*	True	$x.PID = y.AID$, where x : *Process* and y : *Package*
6	*Invoke*	*Stub*	x is remote	$x.R5_O.OID = y.TID$, where x : *Invoke* and y : *Stub*

Each metaclass is encoded as a co-inductive type. For example, consider the following fragment of the encoding of the source metamodel:[4]

```
CoInductive Process : Set :=
   Build_Process (PID: nat) (R1_C: list Class)
with Class : Set :=
   Build_Class (CID: nat) (R1_P : Process) (R2_O: list Operation)
with Operation : Set :=
   Build_Operation (OID: nat) (R2_C: Class) (R3_S: list Statement)
      (R5_I: list Invoke)
...
```

The PO specification of the transformation is then defined in Coq by the following proposition:

```
forall l_p : list Process, exists l_a : list Package,
    forall p : Process,
        p ∈ l_p -> exists a : Package,
            a ∈ l_a /\
            a.AID = p.PID /\
            forall c : Class,
                c ∈ p.R1_C -> forall o : Operation,
                    o ∈ c.R2_O -> forall s : Statement,
                        s ∈ o.R3_S -> equal_P p i.R5_O.R2_C.R1_P = False ->
                            exists t : Stub,
                                t ∈ a.R6_T /\
                                t.TID = i.R5_O.OID .
```

Note that we have simplified the form of the specification, removing references to pre or postconditions if they are $True$.

The proof of the specification proceeds by induction over the lists mentioned in the respective Traverse clauses: l_p, $p'.R1_C$, $c'.R2_O$ and $o'.R3_S$. While the full proof is considerable, besides the careful choice of list induction (suggested by the form of the specification), much of the derivation is achieved automatically in Coq. The extracted model transformation consists of a series of list recursions (corresponding to the uses of induction in the proof) and comes quite close to the kind of transformation a human developer might produce – but with the advantage of being guaranteed correct.

5 Related Work and Conclusions

A number of authors have attempted to provide a formal understanding of metamodelling and model transformations. Ruscio et al. have made some progress

[4] We omit the treatment of generalization here, treating `R4` as a bidirectional relationship between `Statement` and `Invoke`. In our full version, this is treated as a bidirectional relationship to the disjoint union type of `Other` and `Invoke` metaclasses, allowing us to represent the fact that two children can access the attributes of their supertype, but have no access to each others' attributes.

towards formalizing the KM3 metamodelling language using the Abstract State Machines [10]. Rivera and Vallecillo have exploited the class-based nature of the Maude specification language to formalize metamodels written in the KM3 metamodelling language [9]. The intention was to use Maude as a means of defining dynamic behaviour of models, something that our approach also lends itself to. Their work has the advantage of permitting simulation via rewriting rules. A related algebraic approach is given by Boronat and Meseguer in [1]. These formalisms are useful for metamodel verification purposes, but are currently not amenable to the test case generation problem.

Rule-based model transformations (in contrast to procedural/functional ones found in language such as Kermeta and Converge), have a natural formalization in graph rewriting systems [3]. However, their approach is by definition applicable within the rule-based paradigm: in contrast, because our tests are contractual and based in the very generic space of constructive logic, we need not restrict ourselves to rule-based transformations.

We believe that model transformations will never reach their full industrial potential without some guarantee of correctness. This is in contrast to ordinary business programming, where correctness has a benefit-cost ratio that is logarithmic with respect to completeness of guaranteeing proofs. However, a single error in a transformation can have potentially drastic and untraceable effects on code. In terms of benefit-cost ratios, model transformations, while potentially useful throughout all sectors of development, including enterprise computing, should be developed once and verified by experts. We believe that CIC and Coq are a natural choice, as they permit encoding of models, metamodels, transformation specifications and model transformations within a single language and with uniform semantics. For this reason, we see this work as opening up a very promising line of research for the formal metamodelling community. Further details of the case study can be found at

`http://refine-mda.group.shef.ac.uk/sites/default/files/report.pdf`.

References

1. Boronat, A., Meseguer, J.: An algebraic semantics for the MOF. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 377–391. Springer, Heidelberg (2008)
2. Constable, R., Mendler, N., Howe, D.: Implementing Mathematics with the Nuprl Proof Development System. Prentice-Hall, Englewood Cliffs (1986), Updated edition available at `http://www.cs.cornell.edu/Info/Projects/NuPrl/book/doc.html`
3. Königs, A., Schürr, A.: Multi-domain integration with MOF and extended triple graph grammars. In: Bezivin, J., Heckel, R. (eds.) Language Engineering for Model-Driven Software Development number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany (2005)
4. Martin-Löf, P.: An Intuitionstic Theory of Types: Predicate Part. In: Rose, H.E., Shepherdson, J.C. (eds.) Logic Colloquium. North-Holland, Oxford (1973)

5. OMG. Meta Object Facility (MOF) Core Specification, Version 2.0. Object Management Group (January 2006)
6. Poernomo, I.: A type theoretic framework for formal metamodelling. In: Reussner, R., Stafford, J.A., Szyperski, C. (eds.) Architecting Systems with Trustworthy Components. LNCS, vol. 3938, pp. 262–298. Springer, Heidelberg (2006)
7. Poernomo, I.: Proofs-as-model-transformations. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 214–228. Springer, Heidelberg (2008)
8. Poernomo, I., Crossley, J., Wirsing, M.: Adapting Proofs-as-Programs: The Curry-Howard Protocol. Monographs in computer science. Springer, Heidelberg (2005)
9. Rivera, J., Vallecillo, A.: Adding behavioural semantics to models. In: The 11th IEEE International EDOC Conference (EDOC 2007), Annapolis, Maryland, USA, October 15-19, pp. 169–180. IEEE Computer Society, Los Alamitos (2007)
10. Ruscio, D.D., Jouault, F., Kurtev, I., Bézivin, J., Pierantonio, A.: Extending AMMA for supporting dynamic semantics specifications of DSLs. Technical Report 06.02, Laboratoire d'Informatique de Nantes-Atlantique (LINA), Nantes, France (April 2006)

Decision Procedures for the Temporal Verification of Concurrent Lists

Alejandro Sánchez[1] and César Sánchez[1,2]

[1] The IMDEA Software Institute, Madrid, Spain
[2] Spanish Council for Scientific Research (CSIC), Spain
{alejandro.sanchez,cesar.sanchez}@imdea.org

Abstract. This paper studies the problem of formally verifying temporal properties of concurrent datatypes. Concurrent datatypes are implementations of classical data abstractions, specially designed to exploit the parallelism available in multiprocessor architectures. The correctness of concurrent datatypes is essential for the overall correctness of the client software. The main difficulty to reason about concurrent datatypes is due to the simultaneous use of unstructured concurrency and dynamic memory.

The first contribution of this paper is the use of deductive temporal verification methods, in particular verification diagrams, enriched with reasoning about dynamic memory. Proofs using verification diagrams are decomposed into a finite collection of verification conditions. Our second contribution is a decision procedure mixing memory regions, pointers and lisp-like lists with locks, that allows the automatic verification of the generated verification conditions. We illustrate our techniques proving safety and liveness properties of lock-coupling concurrent lists.

1 Introduction

Concurrent data structures [5] are an efficient approach to exploit the parallelism of multiprocessor architectures. In contrast with sequential implementations, concurrent datatypes allow the simultaneous access of many threads to the memory representing the data value of the concurrent datatype. Concurrent data structures are hard to design, difficult to implement correctly and even more difficult to formally prove correct.

The main difficulty in reasoning about concurrent datatypes comes from the interaction of concurrency and heap manipulation. The most popular technique to reason about the structures in the heap is separation logic [10]. Leveraging on this success, some researchers [6,13] have extended this logic to deal with concurrent programs. However, in separation logic disjoint regions are implicitly declared (hidden in the separation conjunction), which makes the reasoning about unstructured concurrency more cumbersome.

In this paper, we propose a complementary approach. We start from temporal deductive verification in the style of Manna-Pnueli [7], in particular using general verification diagrams [4,11] to deal with concurrency. Then, inspired by regional

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 74–89, 2010.
© Springer-Verlag Berlin Heidelberg 2010

logic [1], we enrich the state predicate language to reason about the different regions in the heap that a program manipulates. Finally, we build decision procedures capable of checking all generated verification conditions generated during our proofs, to aid in the automation of the verification process.

Explicit regions allow the use of a classical first-order assertion language to reason about heaps, including mutation and disjointness of memory regions. Regions correspond to finite sets of object references. Unlike separation logic, the theory of sets [14] can be easily combined with other classical theories to build more powerful decision procedures. Classical theories are also amenable of integration into SMT solvers [2]. Moreover, being a classical logic one can use classical Assume-Guarantee reasoning, for example McMillan proof rules [8], for reasoning compositionally about liveness properties. In practice, using explicit regions requires the annotation and manipulation of ghost variables of type *region*, but adding these annotations is usually straightforward.

Verification diagrams can be understood as an intuitive way to abstract the specific aspect of a program which illustrates why the program satisfies a given temporal property We propose the following verification process to show that a datatype satisfies a property expressed in linear temporal logic. First, we build the *most general client* of the datatype, parametrized by the number of threads. Then, we annotate the client and datatype with ghost fields and ghost code to support the reasoning, if necessary. Second, we build a verification diagram that serves as a witness of the proof that all possible parallel executions of the program satisfy the given temporal formula.

The proof is checked in two phases. First, we check that all executions abstracted by the diagram satisfy the property, which can be solved through a fully-automatic finite state model checking method. Second, we must check that the diagram does in fact abstract the program, which reduces to verifying a collection of verification conditions, generated from the diagram. Each concurrent datatype maintains in memory a collection of nodes and pointers with a particular layout. Based on this fact, we propose to use an assertion language whose terms include predicates in specific theories for each layout. For instance, in the case of singly linked lists, we use a decision procedure capable of reasoning about ideal lists as well as pointers representing lists in memory. In this paper, we build a decision procedure extending the theory of linked lists [9] with locks. We illustrate the whole approach to prove thread termination on a simple implementation of concurrent lists.

Most previous approaches to verifying concurrent datatypes are restricted to safety properties. In comparison, the method we propose can be used to prove *all liveness* properties, relying on the completeness of verification diagrams.

The rest of the paper is structured as follows. Section 2 presents the running example: lock-coupling concurrent lists. Section 3 briefly introduces verification diagrams and explicit regions. Section 4 describes the proposed decision procedure for concurrent lists. Finally, Section 5 shows how to apply our approach to prove termination in one case of concurrent lists. Some proofs are missing due to space limitations.

2 Concurrent Lock-Coupling Lists

The running example in this paper is the verification of lock-coupling concurrent lists [5,13]. Lock-coupling concurrent lists are ordered lists with non-repeating elements, in which each node is protected by a lock. A thread advances through the list acquiring the lock of the node it visits. This lock is only released after the lock of the next node has been acquired. The *List* and *Node* structures, shown in Fig. 1(a) are used to maintain the data of a concurrent list.

A *List* contains one field pointing to the *Node* representing the head of the list. A *Node* consists of a value, a pointer to the next *Node* in the list and a lock. We assume that the operating system provides the operations *lock* and *unlock* to acquire and release a lock. Every list has two sentinel nodes, *Head* and *Tail*, with phantom values representing the lowest and highest possible values. For simplicity, we assume such nodes cannot be removed or modified. Concurrent Lock-Coupling Lists are used to implement sets, so they offer three operations:

– *locate*, shown in Fig. 1(d), finds an element traversing the list. This operation returns the pair consisting of the desired node and the node that precedes it in the list. If the element is not found the *Tail* node is returned as the

```
class List {
    Node list;
}

class Node {
    Value val;
    Node next;
    Lock lock;
}
```

(a) data structures

```
1: while true do
2:     e := NondetPickElem
3:     nondet
4:     [ call search(e)
         or
         call add(e)
         or
         call remove(e) ]
5: end while
```

(b) *decide*

```
1: prev, curr := locate(e)
2: if curr.val = e then
3:     result := true
4: else
5:     result := false
6: end if
7: curr.unlock()
8: prev.unlock()
9: return result
```

(c) *search*

```
1: prev := Head
2: prev.lock()
3: curr := prev.next
4: curr.lock()
5: while curr.val < e do
6:     prev.unlock()
7:     prev := curr
8:     curr := curr.next
9:     curr.lock()
10: end while
11: return (prev, curr)
```

(d) *locate*

```
1: prev, curr := locate(e)
2: if curr.val ≠ e then
3:     aux := new Node(e)
4:     aux.next := curr
5:     prev.next := aux
6:     result := true
7: else
8:     result := false
9: end if
10: prev.unlock()
11: curr.unlock()
12: return result
```

(e) *add*

```
1: prev, curr := locate(e)
2: if curr.val = e then
3:     aux := curr.next
4:     prev.next := aux
5:     result := true
6: else
7:     result := false
8: end if
9: prev.unlock()
10: curr.unlock()
11: return result
```

(f) *remove*

Fig. 1. Data structure and algorithms for concurrent lock-coupling list

current node. A search operation, shown in Fig. 1(c), that decides whether an element is in the list can be easily extended from *locate*.

- *add*, shown in Fig. 1(e), inserts a new element in the list, using *locate* to determine the position at which the element must be inserted. The operation *add* returns *true* upon success, otherwise it returns *false*.
- *remove*, in Fig. 1(f), deletes a node from the list by redirecting the next pointer of the previous node appropriately.

Fig. 1(b) shows the most general client of the concurrent-list datatype: the program *decide* that repeatedly chooses non-deterministically a method and its parameters. We construct a fair transition system $\mathcal{S}[N]$ parametrized by the total number of threads N, in which all threads run *decide*. Let ψ be the temporal formula that describes that the thread which holds the last lock in the list terminates. The verification problem is then casted as $\mathcal{S}[N] \vDash \psi$, for all N.

A sketch of a verification diagram is depicted in Fig. 2. We say that a thread is the rightmost owning a lock when there is no other thread owning a lock that protects a *Node* closer to the tail. Each diagram node is labeled with a predicate. This predicate captures the set of states of the transition system that the node abstracts. Edges represent transitions between states abstracted by the nodes.

Checking the proof represented by the verification diagram requires two activities. First, show that all traces of the diagram satisfy the temporal formula ψ, which can be performed by finite state model checking. Second, prove that all computations of $\mathcal{S}[N]$ are traces of the verification diagram. This process involves the verification of formulas built from several theories. For instance, considering the execution of line 5 of program *add* we should verify that the following condition holds:

$$at_add_5^{[k]} \wedge IsLast(k) \wedge \begin{pmatrix} r' = r \cup \langle aux^{[k]} \rangle & \wedge \\ prev'^{[k]}.next = aux^{[k]} & \end{pmatrix} \rightarrow at'_add_6^{[k]} \wedge IsLast'(k)$$

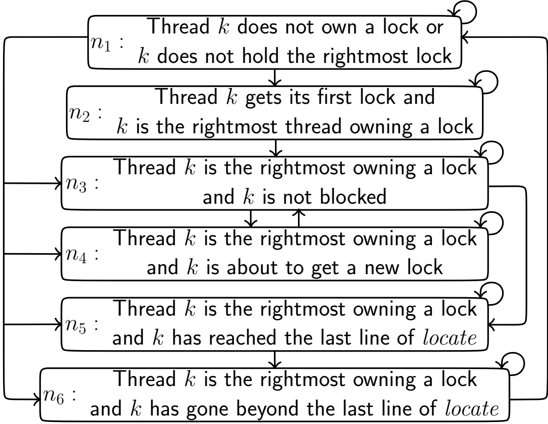


Fig. 2. Sketched verification diagram for $\mathcal{S}[N] \vDash \psi$

The predicate $prev'^{[k]}.next = curr^{[k]}$ is in the theory of pointers, while $r' = r \cup \langle curr^{[k]} \rangle$ is in the theory of regions. Moreover, some predicates belong to a combination of theories, like $IsLast(k)$, which among other things establishes that $List\,(h, x, r)$ holds. $List\,(h, x, r)$ expresses that in heap h, starting from pointer x, the pointers form a list of elements following the *next* field, and that all nodes in this list form precisely the region r.

The construction of a verification diagram is a manual task, but it often follows the programmer's intuitive explanation of why the property holds. The activity that we want to automate is checking that the diagram indeed proofs the property. To accomplish this automation we must build a suitable decision procedure involving many theories, which we describe in the rest of the paper.

3 Preliminaries

We describe the temporal properties of interest in linear temporal logic, using operators such as $\Box$ (always), $\Diamond$ (eventually), $\bigcirc$ (next) or $\mathcal{U}$ (until) in conjunction with classical logic operations. The state predicates are built from the combination of theories that we present here.

Explicit Regions. We use explicit regions to represent the manipulation of memory during the execution of the system. This reasoning is handled by extending the program code with ghost variables of type **rgn**, and ghost updates of these variables. Variables of type **rgn** represent finite sets of object references stored in the heap. Regional logic [1] provides a rich set of language constructs and assertions. However, it is enough for our purposes to use only a small fragment of regional logic. The term **emp** denotes the empty region and $\langle x \rangle$ represents the singleton region whose only object is the one referenced by x. Traditional set-like operators such as $\cup$, $\cap$ and $\setminus$ are also provided and can be applied to **rgn** variables. The assertion language allows reasoning involving mutation and separation. Given two **rgn** expressions r_1 and r_2 we can assert whether they are equal ($r_1 = r_2$), one is contained into the other ($r_1 \subseteq r_2$) or they are completely disjoint ($r_1 \# r_2$).

Verification Diagrams. We sketch here the important notions from [4, 11]. Verification diagrams provide an intuitive way to abstract temporal proofs over fair transition systems (FTS). A FTS Φ is a tuple $\langle \mathcal{V}, \Theta, \mathcal{T}, \mathcal{J} \rangle$ where $\mathcal{V}$ is a finite set of variables, Θ is an initial assertion, $\mathcal{T}$ is a finite set of transitions and $\mathcal{J} \subseteq \mathcal{T}$ contains the fair transitions (in this paper we will not discuss strong fairness). A *state* is an interpretation of $\mathcal{V}$. We use $\mathbf{S}$ to denote the set of all possible states. A transition $\tau \in \mathcal{T}$ is a function $\tau : \mathbf{S} \to 2^{\mathbf{S}}$, which is usually represented by a first-order logic formula $\rho_\tau(s, s')$ describing the relation between the values of the variables in a state s and in a successor state s'. Given a transition τ, the state predicate $En(\tau)$ denotes whether there exists a successor state s' such that $\rho_\tau(s, s')$.

A computation of Φ is an infinite sequence of states such that (a) the first state satisfies Θ; (b) any two consecutive states satisfy ρ_τ for some $\tau \in \mathcal{T}$; (c) for each

$\tau \in \mathcal{J}$, if τ is continuously enabled after some point, then τ is taken infinitely many times. We use $\mathcal{L}(\Phi)$ to denote the set of computations of the FTS Φ. Given a formula φ, $\mathcal{L}(\varphi)$ denotes the set of sequences satisfying φ. A FTS Φ satisfies a temporal formula φ if all computations of Φ satisfy φ, i.e., $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\varphi)$.

A verification diagram (VD) $\Psi : \langle N, N_0, E, \mu, \eta, \mathcal{F}, \Delta, f\rangle$ is a formula automaton with components:

- N is a finite set of nodes.
- $N_0 \subseteq N$ is the set of initial nodes.
- $E \subseteq N \times N$ is a set of edges.
- $\mu : N \to F(V)$ is a labeling function mapping nodes to assertions over V.
- $\eta : E \to 2^{\mathcal{T}}$ is a labeling function assigning sets of transitions to edges.
- $\mathcal{F} \subseteq 2^{E\times E}$ is an edge acceptance set of the form $\{(P_1, R_1), \ldots, (P_m, R_m)\}$.
- $\Delta \subseteq \{\delta | \delta : \mathbf{S} \to \mathcal{D}\}$ is a set of ranking functions from states to a well founded domain $\mathcal{D}$.
- f maps nodes into propositional formulas over atomic subformulas of φ.

If $n \in N$ then we use $\mathit{next}(n)$ to denote the set $\{\tilde{n} \in N | (n, \tilde{n}) \in E\}$ and $\tau(n)$ for $\{\tilde{n} \in \mathit{next}(n) | \tau \in \eta(n, \tilde{n})\}$. For each $(P_j, R_j) \in \mathcal{F}$ and for each $n \in N$, Δ contains a ranking function $\delta_{j,n}$. An infinite sequence of nodes $\pi = n_0, n_1, \ldots$ is a path if $n_0 \in N_0$ and for each $i > 0$, $(n_i, n_{i+1}) \in E$. A path π is accepted if for each pair $(P_j, R_j) \in \mathcal{F}$ some edges of R_j occur infinitely often in π or all edges that occur infinitely often in π are also in P_j. An infinite path π is fair when, for any just transition τ, if τ is enabled on all nodes that appear infinitely often in π then τ is taken infinitely often.

Given a sequence of states $\sigma = s_0, s_1, \ldots$ of Φ, a path $\pi = n_0, n_1, \ldots$ is a trail of σ whenever $s_i \vDash \mu(n_i)$ for all $i \geq 0$. An infinite sequence of states σ is a computation of Ψ whenever there exists an accepting trail of σ such that is also fair. $\mathcal{L}(\Psi)$ is the set of computations of Ψ.

A verification diagram shows that $\Phi \vDash \varphi$ via the inclusions $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$. The map f is used to check $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\varphi)$. To show $\mathcal{L}(\Phi) \subseteq \mathcal{L}(\Psi)$ it is enough to prove the following verification conditions:

- *Initiation:* at least one initial node from N_0 satisfies the initial condition of the fair transition system Φ.
- *Consecution:* for every node $n \in N$ and transition $\tau \in \mathcal{T}$,
$$\mu(n)(s) \wedge \rho_\tau(s, s') \to \mu(\mathit{next}(n))(s').$$
- *Acceptance:* for each $(P_j, R_j) \in \mathcal{F}$, if $(n_1, n_2) \in P_j \setminus R_j$ then
$$\rho_\tau(s, s') \wedge \mu(n_1)(s) \wedge \mu(n_2)(s') \to \delta_{j,n_1}(s) \succeq \delta_{j,n_2}(s')$$
and if $(n_1, n_2) \notin P_j \cup R_j$ then
$$\rho_\tau(s, s') \wedge \mu(n_1)(s) \wedge \mu(n_2)(s') \to \delta_{j,n_1}(s) \succ \delta_{j,n_2}(s')$$
- *Fairness:* For each $e = (n_1, n_2) \in E$ and $\tau \in \eta(e)$:
  1. τ is guaranteed to be enabled in every $\mu(n_1)(s)$.
  2. Any τ-successor of a state satisfying $\mu(n_1)$ satisfies the label of some node in $\tau(n)$.

4 Building a Suitable Decision Procedure

The automatic check of the proof represented by a verification diagram requires decision procedures to verify the generated verification conditions. These decision procedures must deal with formulas containing terms belonging to different theories. In particular, for concurrent lists the decision procedure must reason about pointer data structures with a list layout, regions and locks. To obtain a suitable decision procedure, we extend the Theory of Linked Lists (TLL) [9], a decidable theory including reachability of list-like structures. However, this theory lacks the expressivity to describe locked lists of cells, a fundamental component in our proofs.

We begin with a brief description of the basic notation and concepts. A signature Σ is a triple (S, F, P) where S is a set of sorts, F a set of functions and P a set of predicates. If $\Sigma_1 = (S_1, F_1, P_1)$ and $\Sigma_2 = (S_2, F_2, P_2)$ are two signatures, we define their union $\Sigma_1 \cup \Sigma_2 = (S_1 \cup S_2, F_1 \cup F_2, P_1 \cup P_2)$. Similarly we say that $\Sigma_1 \subseteq \Sigma_2$ when $S_1 \subseteq S_2$, $F_1 \subseteq F_2$ and $P_1 \subseteq P_2$. If $t(\varphi)$ is a term (resp. formula), then we denote with $V_\sigma(t)$ (resp. $V_\sigma(\varphi)$) the set of variables of sort σ occurring in t (resp. φ).

A Σ-interpretation is a map assigning a value to each symbol in Σ. A Σ-structure is a Σ-interpretation over an empty set of variables. A Σ-formula over a set X of variables is satisfiable whenever it is true in some Σ-interpretation over X. Let Ω be an interpretation, $\mathcal{A}$ a Ω-interpretation over a set $V$of variables, $\Sigma \subseteq \Omega$ and $U \subseteq V$. $\mathcal{A}^{\Sigma,U}$ denotes the interpretation obtained from $\mathcal{A}$ restricting it to interpret only the symbols in Σ and the variables in U. We use $\mathcal{A}^{\Sigma}$ to denote $\mathcal{A}^{\Sigma,\emptyset}$. A Σ-theory is a pair $(\Sigma, \mathbf{A})$ where Σ is a signature and $\mathbf{A}$ is a class of Σ-structures. Given a theory $T = (\Sigma, \mathbf{A})$, a T-interpretation is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}^{\Sigma} \in \mathbf{A}$. Given a Σ-theory T, a Σ-formula φ over a set of variables X is T-satisfiable if it is true on a T-interpretation over X.

Formally, the theory of linked lists is defined as $\mathsf{TLL} = (\Sigma_{\mathsf{TLL}}, \mathbf{TLL})$, where

$$\Sigma_{\mathsf{TLL}} := \Sigma_{\mathsf{cell}} \cup \Sigma_{\mathsf{mem}} \cup \Sigma_{\mathsf{Reachability}} \cup \Sigma_{\mathsf{set}} \cup \Sigma_{\mathsf{Bridge}}$$

and $\mathbf{TLL}$ is the class of Σ_{TLL}-structures satisfying the conditions shown in Fig. 4. The sorts, functions and predicates of TLL correspond to the signatures shown in Fig. 3. (Note that Figs. 4 and 3 contain an extended signature and interpretation.) Informally, Σ_{cell} models *cells*, structures containing an element (data), an addresses (pointer) and a lock owner, which represent a node in a linked list. Σ_{mem} models the memory. $\Sigma_{\mathsf{Reachability}}$ models finite sequences of non-repeating addresses, to represent paths. Σ_{set} models sets of addresses. Finally, Σ_{Bridge} is a *bridge theory* containing auxiliary functions. The sort thid contains thread identifiers. The sorts addr, elem and thid are uninterpreted, except that $\oslash : \mathsf{thid}$ is different from all others thread ids. Otherwise, $\Sigma_{\mathsf{addr}} = (\mathsf{addr}, \emptyset, \emptyset)$, $\Sigma_{\mathsf{elem}} = (\mathsf{elem}, \emptyset, \emptyset)$ and $\Sigma_{\mathsf{thid}} = (\mathsf{thid}, \emptyset, \emptyset)$.

We extend TLL into the theory of concurrent single linked lists $\mathsf{TLL3} := (\Sigma_{\mathsf{TLL3}}, \mathbf{TLL3})$, where $\Sigma_{\mathsf{TLL3}} = \Sigma_{\mathsf{TLL}} \cup \Sigma_{\mathsf{setth}} \cup \{lockid, lock, unlock, firstlocked\}$. The sorts, functions and predicates of Σ_{TLL3} are described in Fig. 3. $\mathbf{TLL3}$ is the class of Σ_{TLL3}-structures satisfying the conditions listed in Fig. 4.

Signature	Sorts	Functions	Predicates
Σ_{cell}	cell elem addr thid	$error$: cell $mkcell$: elem × addr × thid → cell $_.data$: cell → elem $_.next$: cell → addr $_.lockid$: cell → thid $_.lock$: cell → thid → cell $_.unlock$: cell → cell	
Σ_{mem}	mem addr cell	$null$: addr $_[_]$: mem × addr → cell upd : mem × addr × cell → mem	
$\Sigma_{\mathsf{Reachability}}$	mem addr path	ϵ : path $[_]$: addr → path	$append$: path × path × path $reach$: mem × addr × addr × path
Σ_{set}	addr set	$\emptyset$: set $\{_\}$: addr → set $\cup, \cap, \setminus$: set × set → set	$\in$: addr × set $\subseteq$: set × set
Σ_{setth}	thid setth	$\emptyset_T$: setth $\{_\}_T$: thid → setth $\cup_T, \cap_T, \setminus_T$: setth × setth → setth	$\in_T$: thid × setth $\subseteq_T$: setth × setth
Σ_{Bridge}	mem addr set path	$path2set$: path → set $addr2set$: mem × addr → set $getp$: mem × addr × addr → path $firstlocked$: mem × path → addr	

Fig. 3. The signature of the TLL3 theory

Definition 1 (Finite Model Property). *Let Σ be a signature, $S_0 \subseteq S$ be a set of sorts, and T be a Σ-theory. T has the finite model property with respect to S_0 if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ such that for each sort $\sigma \in S_0$, $\mathcal{A}_\sigma$ is finite.*

TLL [9] enjoys the finite model property. We now show that TLL3 also has the finite model property with respect to domains elem, addr and thid. Hence, TLL3 is decidable because one can enumerate Σ_{TLL3}-structures up to a certain cardinality. To prove this result, we first extend the set of normalized TLL-literals.

Definition 2 (TLL3-normalized literals). *A TLL3-literal is normalized if it is a flat literal of the form:*

$e_1 \neq e_2$	$a_1 \neq a_2$	
$a = null$	$c = error$	
$c = mkcell(e, a)$	$c = rd(m, a)$	$m_2 = upd(m_1, a, c)$
$s = \{a\}$	$s_1 = s_2 \cup s_3$	$s_1 = s_2 \setminus s_3$
$p_1 \neq p_2$	$p = [a]$	$p_1 = rev(p_2)$
$s = path2set(p)$	$append(p_1, p_2, p_3)$	$\neg append(p_1, p_2, p_3)$
$s = addr2set(m, a)$	$p = getp(m, a_1, a_2)$	
$k_1 \neq k_2$	$c = mkcell\,(e, a, k)$	$a = firstlocked\,(m, p)$

where e, e_1 and e_2 are elem-variables, a, a_1 and a_2 are addr-variables, c is a cell-variable, m, m_1 and m_2 are mem-variables, p, p_1, p_2 and p_3 are path-variables, and k, k_1 and k_2 are thid-variables.

Lemma 1. *Deciding the TLL3-satisfiability of a quantifier-free TLL3-formula is equivalent to verifying the TLL3-satisfiability of the normalized TLL3-literals.*

Interpretation of sort symbols: cell, mem, path, set and setth	
Each sort σ in Σ_{TLL3} is mapped to a non-empty set $\mathcal{A}_\sigma$ such that: (a) $\mathcal{A}_{\mathsf{cell}} = \mathcal{A}_{\mathsf{elem}} \times \mathcal{A}_{\mathsf{addr}} \times \mathcal{A}_{\mathsf{thid}}$ (c) $\mathcal{A}_{\mathsf{path}}$ is the set of all finite sequences of (pairwise) distinct elements of $\mathcal{A}_{\mathsf{addr}}$	(b) $\mathcal{A}_{\mathsf{mem}} = \mathcal{A}_{\mathsf{cell}}^{\mathcal{A}_{\mathsf{addr}}}$ (d) $\mathcal{A}_{\mathsf{set}}$ is the power-set of $\mathcal{A}_{\mathsf{addr}}$ (e) $\mathcal{A}_{\mathsf{setth}}$ is the power-set of $\mathcal{A}_{\mathsf{thid}}$

Signature	Interpretation
Σ_{cell}	– $mkcell(e, a, k) = \langle e, a, k\rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $k \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e, a, t\rangle.data^{\mathcal{A}} = e$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e, a, t\rangle.next^{\mathcal{A}} = a$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e, a, t\rangle.lockid^{\mathcal{A}} = t$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e, a, t\rangle.lock^{\mathcal{A}}(t') = \langle e, a, t'\rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t, t' \in \mathcal{A}_{\mathsf{thid}}$ – $\langle e, a, t\rangle.unlock^{\mathcal{A}} = \langle e, a, \oslash\rangle$ for each $e \in \mathcal{A}_{\mathsf{elem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $t \in \mathcal{A}_{\mathsf{thid}}$ – $error^{\mathcal{A}}.next^{\mathcal{A}} = null^{\mathcal{A}}$
Σ_{mem}	– $m[a]^{\mathcal{A}} = m(a)$ for each $m \in \mathcal{A}_{\mathsf{mem}}$ and $a \in \mathcal{A}_{\mathsf{addr}}$ – $upd^{\mathcal{A}}(m, a, c) = m_{a \mapsto c}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $a \in \mathcal{A}_{\mathsf{addr}}$ and $c \in \mathcal{A}_{\mathsf{cell}}$ – $m^{\mathcal{A}}(null^{\mathcal{A}}) = error^{\mathcal{A}}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$
$\Sigma_{\mathsf{Reachability}}$	– $\epsilon^{\mathcal{A}}$ is the empty sequence – $[i]^{\mathcal{A}}$ is the sequence containing $i \in \mathcal{A}_{\mathsf{addr}}$ as the only element – $([i_1, \dots, i_n], [j_1, \dots, j_m], [i_1, \dots, i_n, j_1, \dots, j_m]) \in append^{\mathcal{A}}$ iff i_k and j_l are all distinct – $(m, i, j, p) \in reach^{\mathcal{A}}$ iff $i = j$ and $p = \epsilon$, or there exist addresses $i_1, \dots, i_n \in \mathcal{A}_{\mathsf{addr}}$ such that: (a) $p = [i_1, \dots, i_n]$ (c) $m(i_r).next^{\mathcal{A}} = i_{r+1}$, for $1 \leq r < n$ (b) $i_1 = i$ (d) $m(i_n).next^{\mathcal{A}} = j$
Σ_{set}	The symbols $\emptyset$, $\{_\}$, $\cup$, $\cap$, $\setminus$, $\in$ and $\subseteq$ are interpreted according to their standard interpretation over sets of addresses.
Σ_{setth}	The symbols $\emptyset_T$, $\{_\}_T$, $\cup_T$, $\cap_T$, $\setminus_T$, $\in_T$ and $\subseteq_T$ are interpreted according to their standard interpretation over sets of thread identifiers.
Σ_{Bridge}	– $addr2set^{\mathcal{A}}(m, i) = \left\{ j \in \mathcal{A}_{\mathsf{addr}} \mid \exists p \in \mathcal{A}_{\mathsf{path}} \text{ s.t. } (m, i, j, p) \in reach \right\}$ – $path2set^{\mathcal{A}}(p) = \{i_1, \dots, i_n\}$ for $p = [i_1, \dots, i_n] \in \mathcal{A}_{\mathsf{path}}$ – $getp^{\mathcal{A}}(m, i, j) = \begin{cases} p & \text{if } (m, i, j, p) \in reach^{\mathcal{A}} \\ \epsilon & \text{otherwise} \end{cases}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$, $p \in \mathcal{A}_{\mathsf{path}}$ and $i, j \in \mathcal{A}_{\mathsf{addr}}$ – $firstlocked^{\mathcal{A}}(m, [a_1, \dots, a_n]) = \begin{cases} a_k & \text{if there is } 1 \leq k \leq n \text{ such that} \\ & \text{for all } 1 \leq j < k, m[a_j].lockid = \oslash \\ & \text{and } m[a_k].lockid \neq \oslash \\ null & \text{otherwise} \end{cases}$ for each $m \in \mathcal{A}_{\mathsf{mem}}$ and $a_1, \dots a_n \in \mathcal{A}_{\mathsf{addr}}$

Fig. 4. Characterization of a TLL3-interpretation $\mathcal{A}$

Proof. By cases on the shape of all possible TLL3-literals. □

Consider an arbitrary TLL3-interpretation $\mathcal{A}$ satisfying a conjunction of normalized TLL3-literals Γ. We show that if there are sets $\mathcal{A}_{\mathsf{elem}}$, $\mathcal{A}_{\mathsf{addr}}$ and $\mathcal{A}_{\mathsf{thid}}$ then there are finite sets $\mathcal{A}'_{\mathsf{elem}}$, $\mathcal{A}'_{\mathsf{addr}}$ and $\mathcal{A}'_{\mathsf{thid}}$ with bounded cardinalities (the bound depending on Γ). $\mathcal{A}'_{\mathsf{elem}}$, $\mathcal{A}'_{\mathsf{addr}}$ and $\mathcal{A}'_{\mathsf{thid}}$ can in turn be used to obtain a finite interpretation $\mathcal{A}'$ satisfying Γ.

Lemma 2 (Finite Model Property). *Let Γ be a conjunction of normalized* TLL3*-literals. Let* $\bar{e} = |V_{\mathsf{elem}}(\Gamma)|$, $\bar{a} = |V_{\mathsf{addr}}(\Gamma)|$, $\bar{m} = |V_{\mathsf{mem}}(\Gamma)|$, $\bar{p} = |V_{\mathsf{path}}(\Gamma)|$ *and* $\bar{k} = |V_{\mathsf{thid}}(\Gamma)|$. *Then the following are equivalent:*

1. *Γ is* TLL3*-satisfiable;*
2. *Γ is true in a* TLL3 *interpretation $\mathcal{A}$ such that*

$$|\mathcal{A}_{\mathsf{elem}}| \leq \bar{e} + \bar{m}\,|\mathcal{A}_{\mathsf{addr}}|$$
$$|\mathcal{A}_{\mathsf{addr}}| \leq \bar{a} + 1 + \bar{m}\,\bar{a} + \bar{p}^2 + \bar{p}^3 + \bar{m}\bar{p}$$
$$|\mathcal{A}_{\mathsf{thid}}| \leq \bar{k} + \bar{m}\,|\mathcal{A}_{\mathsf{addr}}| + 1$$

Proof. $(2 \to 1)$ is immediate. $(1 \to 2)$, by case analysis on normalized TLL3 literals. □

Lemma 2 justifies a brute force method to automatically check TLL3 satisfiability of normalized TLL3-literals. However, such a method is not efficient in practice. To find a more efficient decision procedure we decompose TLL3 into a combination of theories, and apply a many-sorted variant of the Nelson-Oppen combination method [12]. This method requires the theories to fulfill two conditions. First, each theory must have a decision procedure. Second, all involved theories must be stable infinite and share sorts only.

Definition 3 (stable-infiniteness). *A Σ-theory T is stably infinite if for every T-satisfiable quantifier-free Σ-formula φ there exists a T-interpretation $\mathcal{A}$ satisfying φ whose domain is infinite.*

All theories involved in TLL [9] are stably-infinite, so the only missing theory is the one defining *firstlocked*. We define the theory T_{Base3} as follows:

$$T_{\mathsf{Base3}} = T_{\mathsf{addr}} \oplus T_{\mathsf{elem}} \oplus T_{\mathsf{cell}} \oplus T_{\mathsf{mem}} \oplus T_{\mathsf{path}} \oplus T_{\mathsf{set}} \oplus T_{\mathsf{setth}} \oplus T_{\mathsf{thid}}$$

where T_{path} extends the theory of finite sequences of addresses with the auxiliary functions and predicates shown in Fig. 5.

The theory of finite sequences of addresses is defined by $T_{\mathsf{fseq}} = (\Sigma_{\mathsf{fseq}}, \mathsf{TGen})$, where $\Sigma_{\mathsf{fseq}} = \big(\{\mathsf{addr}, \mathsf{fseq}\}, \{\mathit{nil} : \mathsf{fseq}, \mathit{cons} : \mathsf{addr} \times \mathsf{fseq} \to \mathsf{fseq}, \mathit{hd} : \mathsf{fseq} \to \mathsf{addr}, \mathit{tl} : \mathsf{fseq} \to \mathsf{fseq}\}, \emptyset\big)$ and TGen as the class of multi-sorted term-generated structures that satisfy the axioms of T_{fseq}. These axioms are the standard for a theory of lists, such as distinctness, uniqueness and generation of sequences using the constructors *cons* and *nil*, as well as acyclicity of sequences (see, for example [3]). Let *PATH* be the set of axioms of T_{fseq} including all in Fig. 5. Then, we can formally define $T_{\mathsf{path}} = (\Sigma_{\mathsf{path}}, \mathsf{ETGen})$ where ETGen is $\big\{\mathcal{A}^{\Sigma_{\mathsf{path}}} | \mathcal{A}^{\Sigma_{\mathsf{path}}} \models \mathit{PATH} \text{ and } \mathcal{A}^{\Sigma_{\mathsf{fseq}}} \in \mathsf{TGen}\big\}$. Next, we extend T_{Base3} defining the missing functions and predicates from $T_{\mathsf{Reachability}}$ and Σ_{Bridge}. For example:

$$\mathit{ispath}(p) \land \mathit{firstmarked}(m, p, i) \leftrightarrow \mathit{firstlocked}(m, p) = i$$

$app : \mathsf{fseq} \times \mathsf{fseq} \to \mathsf{fseq}$
$app(nil, l) = l$ $app(cons(a, l), l') = cons(a, app(l, l'))$

$fseq2set : \mathsf{fseq} \to \mathsf{set}$
$fseq2set(nil) = \emptyset$ $fseq2set(cons(a, l)) = \{a\} \cup fseq2set(l)$

$ispath : \mathsf{fseq}$
$ispath(nil)$ $ispath(cons(a, nil))$ $\{a\} \not\subseteq fseq2set(l) \wedge ispath(l) \to ispath(cons(a, l))$

$last : \mathsf{fseq} \to \mathsf{addr}$
$last(cons(a, nil)) = a$ $l \neq nil \to last(cons(a, l)) = last(l)$

$isreachable : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr}$
$isreachable(m, a, a)$ $m[a].next = a' \wedge isreachable(m, a', b) \to isreachable(m, a, b)$

$isreachablep : \mathsf{mem} \times \mathsf{addr} \times \mathsf{addr} \times \mathsf{fseq}$
$isreachablep(m, a, a, nil)$ $m[a].next = a' \wedge isreachablep(m, a', b, p) \to isreachablep(m, a, b, cons(a, p))$

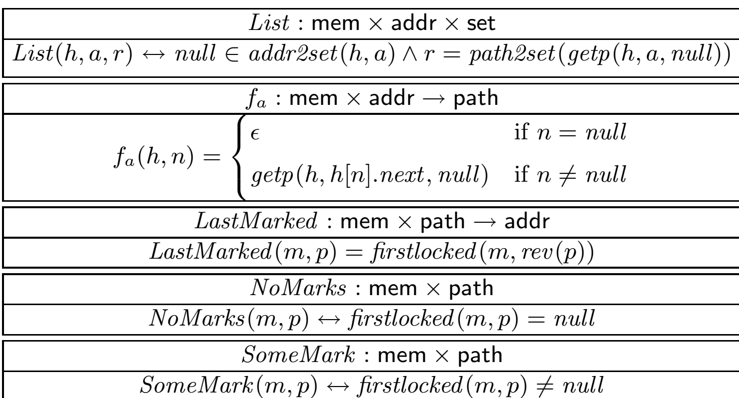
$firstmarked : \mathsf{mem} \times \mathsf{fseq} \times \mathsf{addr}$
$firstmarked(m, nil, null)$ $p \neq nil \wedge p = cons(j, q) \wedge m[j].lockid \neq \oslash \to firstmarked(m, p, j)$ $p \neq nil \wedge p = cons(j, q) \wedge m[j].lockid = \oslash \wedge firstmarked(m, q, i) \to firstmarked(m, p, i)$

Fig. 5. Functions, predicates and axioms of T_{path}

Let GAP be the set of axioms that define ϵ, $[_]$, $append$, $reach$, $path2set$, $addr2set$ and $getp$. We define $\widehat{\mathsf{TLL3}} = (\Sigma_{\widehat{\mathsf{TLL3}}}, \widehat{\mathsf{ETGen}})$ where $\Sigma_{\widehat{\mathsf{TLL3}}}$ is $\Sigma_{\mathsf{TLL}} \cup \{$ $getp$, $append$, $path2set$, $firstlocked$ $\}$ and $\widehat{\mathsf{ETGen}} := \left\{\mathcal{A}^{\Sigma_{\widehat{\mathsf{TLL3}}}} | \mathcal{A}^{\Sigma_{\widehat{\mathsf{TLL3}}}} \models GAP \text{ and } \mathcal{A}^{\Sigma_{\mathsf{path}}} \in \mathsf{ETGen}\right\}$.

Using the definitions of GAP it is easy to prove that if Γ is a set of normalized $\mathsf{TLL3}$-literals, then Γ is $\mathsf{TLL3}$-satisfiable iff Γ is $\widehat{\mathsf{TLL3}}$-satisfiable. Therefore, $\widehat{\mathsf{TLL3}}$ can be used in place of $\mathsf{TLL3}$ for satisfiability checking. We reduce $\widehat{\mathsf{TLL3}}$ into T_{Base3} in two steps. First we do the unfolding of the definition of auxiliary functions defined in $PATH$ and GAP, getting rid of the extra functions, and obtaining a formula in $\widehat{\mathsf{TLL3}}$ and T_{Base}. Then, we use the known reduction [9] from $\widehat{\mathsf{TLL}}$ into T_{Base}. All theories involved in T_{Base3} share only sorts symbols, are stably-infinite and for all of them there is a decision procedure. Hence, the multi-sorted Nelson-Oppen combination method can be applied, obtaining a decision procedure for $\mathsf{TLL3}$.

We now define some auxiliary functions and predicates using $\mathsf{TLL3}$, that aid in the reasoning about concurrent linked-lists (see Fig. 6). For example, predicate $List(h, a, r)$ expresses that in heap h, starting from address a there is sequence of cells all of which form region r. Function $LastMarked(h, p)$, on the other hand, returns the address of the last locked node in path p on memory h. All these



$List$: mem × addr × set
$List(h, a, r) \leftrightarrow null \in addr2set(h, a) \wedge r = path2set(getp(h, a, null))$

f_a : mem × addr → path
$f_a(h,n) = \begin{cases} \epsilon & \text{if } n = null \\ getp(h, h[n].next, null) & \text{if } n \neq null \end{cases}$

$LastMarked$: mem × path → addr
$LastMarked(m, p) = firstlocked(m, rev(p))$

$NoMarks$: mem × path
$NoMarks(m, p) \leftrightarrow firstlocked(m, p) = null$

$SomeMark$: mem × path
$SomeMark(m, p) \leftrightarrow firstlocked(m, p) \neq null$

Fig. 6. Auxiliary functions to reason about concurrent lists

functions can be used in verification conditions. Then, using the equivalences in Fig. 6 the predicates are removed, generating a pure $\widehat{\mathsf{TLL3}}$ formula whose satisfiability can be checked with the procedure described above.

5 Termination of Concurrent Lock-Coupling Lists

In this section we show the proof of a simple liveness property of concurrent lock-coupling lists: termination of the leading thread.

To aid in the verification of this property we annotate the code in Fig. 1 with ghost fields and ghost updates, as shown in Fig. 7, where the boxes represent the annotations introduced. The predicate $c.lockid = \oslash$ denotes that the lock of list node c is not taken. The predicate $c.lockid = k$ establishes that the lock at list node c is owned by thread k. We enrich $List$ objects with a ghost field r of type region that keeps track of all the nodes in the list. The code for add and $remove$ is extended with ghost updates to maintain r.

T_k denotes thread k. We want to prove that if a thread has acquired a lock at node n and no other thread holds a lock ahead of n, then thread k eventually terminates. The predicate $at_add_n^{[k]}$ means that thread k is executing line n of program add. Similarly, $at_add_{n_1,\ldots,n_m}^{[k]}$ is a short for thread k is running some of the lines $n_1, \ldots, n_m$ of program add. To reduce notation, $\tau_{a_n}^{[k]}$, $\tau_{r_n}^{[k]}$ and $\tau_{l_n}^{[k]}$ denote $\tau_{add_n}^{[k]}$, $\tau_{remove_n}^{[k]}$ and $\tau_{locate_n}^{[k]}$ respectively. The instance of a local variable v in thread k is represented by $v^{[k]}$. We define $DisjList$ as an extension of $List$ enriching it with the property that new nodes created during insertion are all disjoint one from each other, including all nodes that are already part of the list:

$$\begin{aligned} DisjList(h, a, r) \mathrel{\hat{=}} List(h, a, r) \wedge \forall j : T_{ID}.at_a_{4,5}^{[j]} \rightarrow \langle aux^{[j]} \rangle \# r \wedge \\ \forall i, j : T_{ID}.i \neq j \wedge at_a_{4,5}^{[i]} \wedge at_a_{4,5}^{[j]} \rightarrow \langle aux^{[i]} \rangle \# \langle aux^{[j]} \rangle \# r \end{aligned}$$

We now define the following auxiliary predicate:

$$IsLast(k) \mathrel{\hat{=}} DisjList(h, l.list, l.r) \wedge SomeMark\big(h, getp(h, l.list, null)\big) \wedge LastMarked\big(h, getp(h, l.list, null)\big) = a \quad \wedge \quad h[a].lockid = k$$

The formula $IsLast(k)$ identifies whether T_k is the thread owning the last lock in the list (i.e., the closest node towards the end of the list). Using these predicates we define the parametrized temporal formula we want to verify as:

$$\psi(k) \mathrel{\hat{=}} \Box\left(at_locate^{[k]}_{3..10} \wedge IsLast(k) \rightarrow IsLast(k)\,\mathcal{U}\; at_locate^{[k]}_{11}\right)$$

This temporal formula states that if thread k is running *locate* and it owns the last locked node in the list, then thread T_k will still own the last locked node until T_k reaches the last line of *locate*. Reachability of the last line of *locate* implies termination of the invocation to the concurrent datatype because *locate* is the only program containing potentially blocking operations.

We proceed with the construction of a verification diagram that proves the parallel execution of all threads guarantee the satisfaction of formula $\psi(k)$. Given N, we build the transitions system $\mathcal{S}[N]$, in which threads $T_1, \ldots, T_N$ run in parallel the program *decide* and show that $\mathcal{S}[N] \vDash \psi(k)$. The verification diagram

```
class List {
    Node list;
    [ rgn r; ]
}

class Node {
    Value val;
    Node next;
    Lock lock;
}
```

(a) data structure

```
1:  prev := Head
2:  prev.lock()
3:  curr := prev.next
4:  curr.lock()
5:  while curr.val < e do
6:      prev.unlock()
7:      prev := curr
8:      curr := curr.next
9:      curr.lock()
10: end while
11: return (prev, curr)
```

(b) *locate*

```
1:  prev, curr := locate(e)
2:  if curr.val ≠ e then
3:      aux := new Node(e)
4:      aux.next := curr
5:      prev.next := aux
            [ l.r := l.r ∪ ⟨aux⟩ ]
6:      result := true
7:  else
8:      result := false
9:  end if
10: prev.unlock()
11: curr.unlock()
12: return result
```

(c) *add*

```
1:  prev, curr := locate(e)
2:  if curr.val = e then
3:      aux := curr.next
4:      prev.next := aux
            [ l.r := l.r − ⟨curr⟩ ]
5:      result := true
6:  else
7:      result := false
8:  end if
9:  prev.unlock()
10: curr.unlock()
11: return result
```

(d) *remove*

Fig. 7. Concurrent lock-coupling list extended with ghost fields

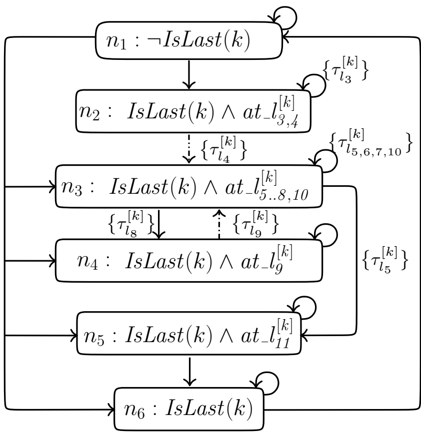


Fig. 8. Verification diagram Ψ for $\|_{j<N} T_j \vDash \psi(k)$

is depicted in Fig. 8. Dashed arrows in the diagram denote transitions that strictly decrement the ranking function δ. Formally, the verification diagram is:

- $N_0 = \{n_1\}$
- $\mathcal{F} = \{(P, R)\}$ where
  $P = \{(n_3, n_4), (n_3, n_5), (n_5, n_6), (n_6, n_1)\} \cup$
  $\qquad \{(n_1, n_j) | j \in 2..6\} \cup \{(n_j, n_j) | j \in 1..6\}$
  $R = \emptyset$
- $\delta(n, s) = \begin{cases} \{a \mid a \in dom(h)\} & n = n_1, n_2 \\ path2set\left(f_a(h, LastMarked(h, getp(h, prev^{[k]}, null)))\right) & \text{otherwise} \end{cases}$
- $f(n) = \begin{cases} \emptyset & \text{if } n = n_1, n_6 \quad at_locate^{[k]}_{3,4} \text{ if } n = n_2 \\ at_locate^{[k]}_{5..8,10} & \text{if } n = n_3 \quad at_locate^{[k]}_{9} \text{ if } n = n_4 \\ at_locate^{[k]}_{11} & \text{if } n = n_5 \end{cases}$

We can now describe the verification conditions:

initialization. Trivial, since in the initial state *l.list* forms an empty list, and consequently $\neg IsLast(k)$.

consecution. We will show, for illustration purposes, transition $\tau^{[j]}_{l_9}$ on node n_2 with $j \neq k$. The verification condition is:

$$\left(\begin{array}{c} \overbrace{IsLast(k)}^{\mathsf{TLL3}} \wedge \overbrace{j \neq k}^{T_{\mathsf{thid}}} \wedge \\ at_l^{[k]}_{3,4} \wedge at_l^{[j]}_{9} \wedge \\ \underbrace{curr^{[j]}.lockid = \oslash}_{\mathsf{TLL3}} \end{array}\right) \wedge \overbrace{curr^{[j]}.lock(j)}^{\mathsf{TLL3}} \rightarrow \left(\begin{array}{c} \overbrace{IsLast(k')}^{\mathsf{TLL3}} \wedge at'_l^{[k']}_{3,4} \wedge \\ at'_l^{[j']}_{10} \wedge pres(\mathcal{V} - curr^{[j]}) \wedge \\ \underbrace{curr'^{[j']}.lockid = j'}_{\mathsf{TLL3}} \end{array}\right)$$

where *pres* is the predicate denoting variable preservation. Note that all fragments of such verification condition belong to theories for which we have

already defined a decision procedure, including propositional logic for the (finite) locations of the program counters.

acceptance. The ranking function δ maps, at a given state, the set of list nodes accessible from the last node with an owned lock. This set remains identical for all transitions except $\tau_{l_4}^{[k]}$ and $\tau_{l_9}^{[k]}$, for which the set decrements (in the inclusion order on sets). The decision procedure presented in Section 4 proves this automatically (using $\subset$ operation and equality over sets of addresses).

fairness. Only two conditions must be verified. First, all transitions labeling an edge are enabled since the only potentially blocking operation is $\tau_{l_9}^{[k]}$ and $\mathit{IsLast}(k)$ implies that $\tau_{l_9}^{[k]}$ is enabled. Second, for all nodes and labelled edges, starting from a state that satisfies the predicate of the incoming node satisfies the predicate of the outgoing node via taking the transition. Sequential progress of thread k is guaranteed by fairness, since all idling transitions for thread k are in fact a diagram idiom to represent the expansion of such nodes to a sequence of nodes with a single program position on each node.

satisfaction. $\mathcal{L}(\Psi) \subseteq \mathcal{L}(\psi(k))$ is automatically checkable via a finite LTL model-checking problem.

6 Conclusion

We have presented a method for the verification of temporal properties (safety and liveness) of an imperative implementation of concurrent lists. The verification is performed using verification diagrams – a complete method to prove temporal properties of reactive systems – and explicit reasoning of memory regions. The verification process usually requires the aid of ghost variables. Checking a proof is reduced to proving a finite number of verification conditions, which requires decision procedures in the appropriate theories, including regions, pointers, locks and specific theories for memory layouts, in this case single linked-lists. This paper also presents a decision procedure built as a combination of theories.

There are some key differences with other approaches in the literature. Building on the success of separation logic in proving sequential programs, the most popular approach has been extending separation logic to concurrent programs. These extensions require adapting techniques like rely-guarantee that cannot be directly used with separation logic. Our decision to use explicit regions (finite sets of addresses) allows the direct use of classical techniques like assume-guarantee and the combination of decision procedures. Furthermore, in concurrent separation logic, it is critical to describe memory footprints of sections of code. This description becomes very cumbersome when the code is not organized in mutual exclusion regions, as in fine-grain synchronization algorithms. Moreover, the integration into SMT solvers is quite straightforward with classical logics, but it is still an open question with separation logic.

The technique we propose can be seen as a method to separate the reasoning about concurrency (with verification diagrams) from the reasoning about the memory (with decision procedures). The former is independent of the data structure under consideration. We are currently extending our approach to the

verification of other pointer-based concurrent data structures like skip-lists or concurrent hash maps. Again, the sharing of these data structures makes it very hard to reason using separation logic. For our approach, these extensions will require the design of suitable decision procedures. Future work also includes building a generic VCgen for verification diagrams, implementing an ad-hoc version of the decision procedure described here, and later integrating this decision procedure into state-of-the-art SMT solvers.

Acknowledgment

We are grateful to the anonymous reviewers for their detailed comments and suggestions.

References

1. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
2. Barrett, C., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability Modulo Theories. In: Handbook of Satifiability. IOS Press, Amsterdam (2008)
3. Bradley, A.R., Manna, Z.: The Calculus of Computation. Springer, Heidelberg (2007)
4. Browne, A., Manna, Z., Sipma, H.B.: Generalized verification diagrams. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 484–498. Springer, Heidelberg (1995)
5. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan-Kaufmann, San Francisco (2008)
6. Hobor, A., Appel, A.W., Nardelli, F.Z.: Oracle semantics for concurrent separation logic. In: Drossopoulou, S. (ed.) ESOP 2008. LNCS, vol. 4960, pp. 353–367. Springer, Heidelberg (2008)
7. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems. Springer, Heidelberg (1995)
8. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–345. Springer, Heidelberg (1999)
9. Ranise, S., Zarba, C.G.: A theory of singly-linked lists and its extensible decision procedure. In: Proc. of Software Engineering and Formal Methods (SEFM 2006). IEEE Computer Society Press, Los Alamitos (2006)
10. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. Logic in Computer Science (LICS 2002), pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
11. Sipma, H.B.: Diagram-Based Verification of Discrete, Real-Time and Hybrid Systems. Ph.D. thesis, Stanford University (1999)
12. Tinelli, C., Zarba, C.G.: Combining decision procedures for sorted theories. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 641–653. Springer, Heidelberg (2004)
13. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: Principles & Practice of Parallel Programming (PPOPP 2006), pp. 129–136. ACM, New York (2006)
14. Wies, T., Piskac, R., Kuncak, V.: Combining theories with shared set operations. In: Ghilardi, S. (ed.) FroCoS 2009. LNCS, vol. 5749, pp. 366–382. Springer, Heidelberg (2009)

An Improved Decision Procedure for Propositional Projection Temporal Logic*

Zhenhua Duan and Cong Tian**

Institute of Computing Theory & Technology and ISN Laboratory
Xidian University, Xi'an, 710071, China
{zhhduan, ctian}@mail.xidian.edu.cn

Abstract. A new decision procedure for Propositional Projection Temporal Logic (PPTL) is proposed which is an improvement to the decision procedure given in [4]. The main contribution of the paper is as follows: (1) the relationship between paths in the NFG of a formula R and its models is established and proved; (2) a new Labeled NFG (LNFG) with a set of labels (propositions) is defined; (3) given a formula R, an LNFG of R can be generated by the new decision algorithm, and all models of R can be found; (4) based on the new decision procedure, an improved model checking algorithm is presented and implemented.

Keywords: Propositional Projection Temporal Logic, Satisfiability, Decision Procedure, Model Checking, Verification.

1 Introduction

Propositional Projection Temporal Logic (PPTL) [3] is an extension of Propositional Interval Temporal Logic (PITL) [2]. It is a useful logic for specification and verification of concurrent systems. The advantages of using PPTL are in three folds: (1) PPTL has the expressiveness of full regular expressions [12]. (2) Intervals are useful in specifying state sensitive properties. For example, p holding at the 6th state can be described by $len(6); p$, and $len(10); len(6) \wedge \Diamond p; true$ means p holds between the 10th and 16th states. It is cumbersome to specify these properties by Propositional Linear Temporal Logic (PLTL) [11], Computation Tree Logic (CTL) [10] and their variations. (3) Chop and projection constructs are useful in the specification of sequential and iterative behaviors respectively. These properties cannot (or with difficulty) be described by PLTL or CTL.

Decidability of temporal logics is a fundamental issue in verification, especially model checking. Therefore, it is important to investigate the decidability of PPTL and PITL so that model checking PPTL and PITL can be done. Bowman and Thompson presented a tableaux-based decision procedure for PITL over finite models in 2003 [8]. Later in [4], a decision procedure for PPTL with infinite models was given and the complexity was proved to be non-elementary [13]. The decision procedure can easily

* This research is supported by the NSFC Grant No. 61003078, 60433010, 60873018 and 60910004, National Program on Key Basic Research Project of China (973 Program) Grant No.2010CB328102 and SRFDP Grant 200807010012.

** Corresponding author.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 90–105, 2010.
© Springer-Verlag Berlin Heidelberg 2010

be implemented and also applied to PITL with minor modification. With this decision procedure, normal form, Normal Form Graph (NFG) and Labeled NFG (LNFG) play important roles for constructing models of a formula. Generally, given a formula P, all models of P are contained in its NFG, and determined by its LNFG. That is, an NFG can be treated as an automaton structure while an LNFG can be viewed as an omega automaton with a specified accepting condition. The decision procedure works well for most of PPTL formulas. Nevertheless, accepting conditions on LNFG was only simply expressed in an informal way. When further implementing the decision procedure, we found that to precisely define accepting conditions is neither a trivial nor an intuitive work. What is more, only by the unprecise accepting condition, some formulas cannot be handled in a proper way. For instance, the LNFG of formula, $q \wedge (\bigcirc empty; q) \wedge \Box(p \wedge \bigcirc\bigcirc empty; q)$, can be constructed by the old algorithm as depicted in Fig.1. By the accepting condition, the formula is unsatisfiable. However, in fact, the formula is satisfiable since the finiteness of chop formula $\bigcirc empty; q$ can be satisfied.

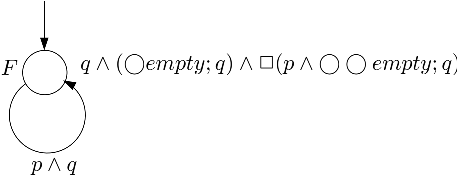


Fig. 1. LNFG of $q \wedge (\bigcirc empty; q) \wedge \Box(p \wedge \bigcirc\bigcirc empty; q)$

The problem is caused by the case in which a circle path occurs, and the circle is composed of nodes (formulas) involving different chop formulas but labeled by the same mark F in the LNFG. To deal with this kind of formulas, only one label F is not sufficient. As a result, we further give an improved decision procedure by means of using distinguishing labels and defining formal accepting conditions in this paper. The improvements focus on four folds: (1) chop formulas and chop components of PPTL formulas are formally defined; (2) finite satisfactory condition, FSC_Property, over infinite paths in the NFG of a formula is explicitly defined to capture the essential property dwelling in the chop construct; (3) models of a formula are precisely specified in its NFG by applying FSC_Property; (4) finally, LNFG of a formula with l_k labels is constructed to explicitly illustrate whether or not FSC_Properties are satisfied over infinite paths. By the above improvements, an improved decision procedure for PPTL with infinite models are formalized. This decision procedure enables us to handle not only basic PPTL formulas but also extended projection star construct. Further, based on the new decision procedure, an improved model checking algorithm to the one presented in [17] is also proposed and implemented. A model checker based on SPIN has been developed recently. By our experience, the decision procedure and the model checking algorithm work well. In addition to verification, the decision procedure is also significant in the theory concerning complementing infinite objections, which is a hard issue [15], since complementing infinite words is implicitly involved in the decision procedure.

The paper is organized as follows. The next section briefly presents the preliminaries, including the syntax and semantics of the underlying logic and the definition of normal form. In section 3, normal form graph is presented in details. Further, the enhanced decision procedure is illustrated in section 4. In section 5, based on the improved decision procedure, an improved model checking algorithm is also presented. Finally, conclusions are drawn in section 6.

2 Preliminaries

2.1 Propositional Projection Temporal Logic

Let *Prop* be a countable set of atomic propositions. The formula P of PPTL is given by the following grammar:

$$P \ ::= \ p \mid \bigcirc P \mid \neg P \mid P_1 \vee P_2 \mid (P_1, ..., P_m) \ prj \ P \mid (P_1, ..., (P_i, ..., P_j)^{\circledast}, ..., P_m) \ prj \ Q$$

where $p \in Prop$, $P_1, ..., P_m$, P and Q are all well-formed PPTL formulas. $\bigcirc$ (*next*), prj (*projection*) and $prj^{\circledast}$ (*projection star*) are basic temporal operators.

Following the definition of Kripke's 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 $s[p]$ to denote the valuation of p at state s. An interval σ is a non-empty sequence of states, which can be finite or infinite. The length, $|\sigma|$, of σ is ω if σ is infinite, and the number of states minus 1 if σ is finite. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider 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 $\leq -\{(\omega, \omega)\}$. To simplify definitions, we will denote σ by $\langle s_0, ..., s_{|\sigma|}\rangle$, where $s_{|\sigma|}$ is undefined if σ is infinite. With such a notation, $\sigma_{(i..j)}$ $(0 \leq i \preceq j \leq |\sigma|)$ denotes the sub-interval $\langle s_i, ..., s_j\rangle$, $\sigma^{(k)}$ $(0 \leq k \preceq |\sigma|)$ denotes $\langle s_k, ..., s_{|\sigma|}\rangle$, i.e., k^{th} suffix of σ, and σ^k $(0 \leq k \preceq |\sigma|)$ denotes $\langle s_0, ..., s_k\rangle$, i.e., k^{th} prefix of σ. Note that for an infinite interval σ, the ωth prefix σ^{ω} and suffix $\sigma^{(\omega)}$ are undefined and denoted by $\perp$. Further, the concatenation of a finite σ with another interval (or empty string) σ' denoted by $\sigma \cdot \sigma'$ is defined as follows.

$$\sigma_1 \cdot \sigma_2 = \begin{cases} \sigma_1 & \text{if } \sigma_2 = \epsilon \\ \sigma_2 & \text{if } \sigma_1 = \epsilon \\ \langle s_0, ..., s_i, s_{i+1}, ...\rangle & \text{if } \sigma_1 = \langle s_0, ..., s_i\rangle \text{ and } \sigma_2 = \langle s_{i+1}, ...\rangle \text{ and } i \in N_0 \\ \perp & \text{otherwise} \end{cases}$$

Let $\sigma = \langle s_0, s_1, \ldots, s_{|\sigma|}\rangle$ be an interval and $r_1, \ldots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the interval (namely projected interval)

$$\sigma \downarrow (r_1, \ldots, r_h) = \langle s_{t_1}, s_{t_2}, \ldots, s_{t_l}\rangle$$

where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. That is, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. For instance,

$$\langle s_0, s_1, s_2, s_3, s_4\rangle \downarrow (0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3\rangle$$

This is convenient to define an interval obtained by taking the endpoints (rendezvous points) of the intervals over which $P_1, \ldots, P_m$ are interpreted in the projection construct.

We need also to generalize the notation of $\sigma \downarrow (r_1, ..., r_h)$ to allow r_h to be ω. For an interval $\sigma =< s_0, s_1, ..., s_{|\sigma|} >$ and $0 \leq r_1 \leq r_2 \leq ... \leq r_h \leq |\sigma|$, we define

$$\sigma \downarrow (r_1, ..., r_{h-1}, r_h) = \sigma \downarrow (r_1, ..., r_{h-1}, r_h) \text{ if } r_h \text{ is not } \omega$$
$$\sigma \downarrow (r_1, ..., r_{h-1}, r_h) = \sigma \downarrow (r_1, ..., r_{h-1}) \text{ if } r_h \text{ is } \omega$$

For instance,

$$< s_0, ... >\downarrow (0, 1, 3, 4, \omega) =< s_0, s_1, s_3, s_4 >$$

An interpretation is a tuple $\mathcal{I} = (\sigma, k, j)$, where σ is an interval, k is an integer, and j an integer or ω such that $k \leq j \leq |\sigma|$. We use the notation $(\sigma, k, j) \models P$ to denote that formula P is interpreted and satisfied over the subinterval $\langle s_k, ..., s_j \rangle$ of σ with the current state being s_k. The satisfaction relation ($\models$) is inductively defined as follows:

I – prop $\mathcal{I} \models p$ iff $s_k[p] = true$, for any given proposition p
I – not $\mathcal{I} \models \neg P$ iff $\mathcal{I} \not\models P$
I – or $\mathcal{I} \models P \vee Q$ iff $\mathcal{I} \models P$ or $\mathcal{I} \models Q$
I – next $\mathcal{I} \models \bigcirc P$ iff $k < j$ and $(\sigma, k+1, j) \models P$
I – prj $\mathcal{I} \models (P_1, ..., P_m)\ prj\ Q$ if there exist integers $k = r_0 \leq r_1 \leq ... \leq r_m \leq j$ such that $(\sigma, r_0, r_1) \models P_1$, $(\sigma, r_{l-1}, r_l) \models P_l, 1 < l \leq m$, and $(\sigma', 0, |\sigma'|) \models Q$ for one of the following σ' :
(a) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, ..., r_m) \cdot \sigma_{(r_m+1..j)}$ or
(b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, ..., r_h)$ for some $0 \leq h \leq m$
I – prj$^{\circledast}$ $\mathcal{I} \models (P_1, ..., (P_i, ..., P_s)^{\circledast}, ..., P_m)\ prj\ Q$ iff :
$1 \leq i \leq s \leq m$ and $\exists n \in N_0, \mathcal{I} \models (P_1, ..., (P_i, ..., P_s)^{(n)}, ..., P_m)\ prj\ Q$
or: $s = m$ and there exist infinitely many integers $k = r_0 \leq r_1 \leq ... \leq r_h \leq \omega$, such that
$(\sigma, r_{l-1}, r_l) \models P_l, 0 < l < i$,
$(\sigma, r_{l-1}, r_l) \models P_t, l \geq i, t = i + ((l-i)\ mod\ (s-i+1))$,
and $(\sigma', 0, |\sigma'|) \models Q$, where $\sigma' = \sigma \downarrow (r_0, r_1, ...)$.

Fig.2 shows the possible semantics of $(P_1, P_2)\ prj\ Q$. Here Q and P_1 start to be interpreted at state t_0; subsequently, P_1 and P_2 are interpreted sequentially; Q is interpreted in parallel with $(P_1; P_2)$ over the interval consisting of endpoints of subintervals over which P_1 and P_2 are interpreted. The possible three cases are given: (a) P_2 terminates before Q; (b) Q and P_2 terminate at the same state; (c) Q terminates before P_2. Projection construction is useful in the specification of concurrent system.

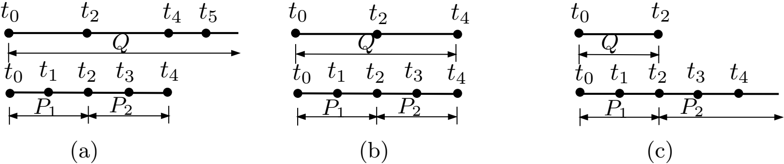


Fig. 2. Semantics of $(P_1, P_2)\ prj\ Q$

In order to avoid an excessive number of parentheses, the following precedence rules are used as shown in Table 1, where 1 = highest and 5 = lowest.

Table 1. Precedence Rules

1 $\neg$	2 $\bigcirc$, $\odot$, $\Diamond$, $\Box$	3 $\wedge$, $\vee$
4 $\rightarrow$, $\leftrightarrow$	5 prj, ;	

The abbreviations $true$, $false$, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $true \stackrel{\text{def}}{=} P \vee \neg P$ and $false \stackrel{\text{def}}{=} P \wedge \neg P$. Also we have the following derived formulas,

$$
\begin{array}{rclrcl}
empty & \stackrel{\text{def}}{=} & \neg \bigcirc true & more & \stackrel{\text{def}}{=} & \neg empty \\
\bigcirc^0 P & \stackrel{\text{def}}{=} & P & \bigcirc^n P & \stackrel{\text{def}}{=} & \bigcirc(\bigcirc^{n-1} P) \\
len(0) & \stackrel{\text{def}}{=} & empty & len(n) & \stackrel{\text{def}}{=} & \bigcirc len(n-1), n \geq 1 \\
skip & \stackrel{\text{def}}{=} & len(1) & \odot P & \stackrel{\text{def}}{=} & empty \vee \bigcirc P \\
P; Q & \stackrel{\text{def}}{=} & (P, Q)\ prj\ empty & \Diamond P & \stackrel{\text{def}}{=} & true; P \\
\Box P & \stackrel{\text{def}}{=} & \neg \Diamond \neg P & keep(P) & \stackrel{\text{def}}{=} & \Box(\neg empty \rightarrow P) \\
halt(P) & \stackrel{\text{def}}{=} & \Box(empty \leftrightarrow P) & fin(P) & \stackrel{\text{def}}{=} & \Box(empty \rightarrow P) \\
P^* & \stackrel{\text{def}}{=} & (P^{\circledast})\ prj\ empty & P^+ & \stackrel{\text{def}}{=} & P; P^*
\end{array}
$$

where $n \geq 1$, $\odot$ (weak next), $\Box$ (always), $\Diamond$ (sometimes), ; (chop) and + (plus) are derived temporal operators; *empty* denotes an interval with zero length, and *more* means the current state is not the final one over an interval; $halt(P)$ is true over an interval if and only P is true at the final state, $fin(P)$ is true as long as P is true at the final state and $keep(P)$ is true if P is true at every state ignoring the final one.

A formula P is satisfied by an interval σ, denoted by $\sigma \models P$, if $(\sigma, 0, |\sigma|) \models P$. A formula P is called satisfiable if $\sigma \models P$ for some σ. A formula P is valid, denoted by $\models P$, if $\sigma \models P$ for all σ.

In some circumstances, an undefined interval may be involved in the interpretations. Let Σ be the set of all intervals, and $\Sigma_\perp = \Sigma \cup \{\perp\}$, where $\perp$ is an undefined interval. We can extend the satisfaction relation to $\Sigma_\perp$. For an undefined interval $\perp$, and any formula P, we define $\perp \models P$.

Theorem 1. Let P be a PPTL formula. We have,

(1) $fin(P) \equiv P \wedge empty \vee \bigcirc fin(P)$
(2) $keep(P) \equiv empty \vee P \wedge \bigcirc keep(P)$
(3) $halt(P) \equiv P \wedge empty \vee \neg P \wedge \bigcirc halt(P)$ □

2.2 Normal Form of PPTL

Normal form is an important notation for constructing NFGs of PPTL formulas. In the following, we briefly give its definition and relevant concepts. The detailed explanation can be found in [4].

Definition 1 (Normal Form). Let Q_p be the set of atomic propositions appearing in a PPTL formula Q. The normal form of Q can be defined by,

$$Q \equiv \bigvee_{j=0}^{n_0}(Q_{ej} \wedge empty) \vee \bigvee_{i=0}^{n_1}(Q_{ci} \wedge \bigcirc Q'_i)$$

where $Q_{ej} \equiv \bigwedge_{k=1}^{m_0} \dot{q}_{jk}$, $Q_{ci} \equiv \bigwedge_{h=1}^{m} \dot{q}_{ih}$, $q_{jk}, q_{ih} \in Q_p$, for any $r \in Q_p$, $\dot{r}$ denotes r or $\neg r$; Q'_i is a PPTL formula without " $\vee$ " being the main operator. □

According to the definition, in a normal form, $p \wedge q \wedge \bigcirc(\Box p \vee q)$ must be written as $p \wedge q \wedge \bigcirc(\Box p) \vee p \wedge q \wedge \bigcirc q$ since " $\vee$ " is the main operator of $\Box p \vee q$. Implicitly, Q'_i is also not permitted to be the form of $\neg \bigwedge_{k=1}^{n\geq 2} P_k$. Further, for convenience, we call $\bigvee_{j=0}^{n_0}(Q_{ej} \wedge empty)$ the terminating part whereas $\bigvee_{i=0}^{n_1}(Q_{ci} \wedge \bigcirc Q'_i)$ the non-terminating part of the normal form.

Definition 2 (Complete Normal Form). Let Q_p be the set of atomic propositions appearing in a PPTL formula Q. The complete normal form of Q is defined by,

$$Q \equiv \bigvee_{j=0}^{n_0}(Q_{ej} \wedge empty) \vee \bigvee_{i=0}^{n_1}(Q_{ci} \wedge \bigcirc Q'_i)$$

where like in the normal form, $Q_{ej} \equiv \bigwedge_{k=1}^{m_0} \dot{q}_{jk}$, $Q_{ci} \equiv \bigwedge_{h=1}^{m} \dot{q}_{ih}$, $q_{jk}, q_{ih} \in Q_p$, for any $r \in Q_p$, $\dot{r}$ denotes r or $\neg r$; further $\bigvee_i Q_{ci} \equiv true$ and $\bigvee_{i\neq j}(Q_{ci} \wedge Q_{cj}) \equiv false$; Q'_i is an arbitrary PPTL formula. □

Note that a complete normal form may be not a normal form, since "$\vee$" is possibly the main operator of Q'_i.

Notice that if Q is transformed into complete normal form, $Q \equiv \bigvee_{j=0}^{n_0}(Q_{ej} \wedge empty) \vee \bigvee_{i=0}^{n_1}(Q_{ci} \wedge \bigcirc Q'_i)$, then $\neg Q$ can be transformed into its normal form, $\neg Q \equiv \bigwedge_{j=0}^{n_0} \neg Q_{ej} \wedge empty \vee \bigvee_{i=0}^{n_1}(Q_{ci} \wedge \bigcirc \neg Q'_i)$. Therefore, it is useful in transforming negation constructs into their normal forms. An important conclusion is that any PPTL formula can be transformed into its normal form [4].

3 Normal Form Graph

Normal Form Graph (NFG) was introduced in [4] for the purpose of obtaining models of PPTL formulas. For convenience, here we also briefly introduce its definition. For a PPTL formula P, NFG of P is a directed graph, $G = (CL(P), EL(P), V_0)$, where $CL(P)$ denotes the set of nodes, $EL(P)$ the set of edges , and $V_0 \subseteq CL(P)$ the set of root nodes in the graph. In $CL(P)$, each node is specified by a formula in PPTL, while in $EL(P)$, each edge is a directed arc, labeled with a state formula Q_e, from node Q to node R and identified by a triple, (Q, Q_e, R). Accordingly, for convenience sometimes, a node in an NFG or LNFG is called a formula. NFG of a PPTL formula is inductively defined in Definition 3.

Definition 3 (Normal Form Graph, NFG). For a PPTL formula P, the set $CL(P)$ of nodes and $EL(P)$ of edges connecting nodes in $CL(P)$ are inductively defined as follows:

1. If P is not in the form of $\bigvee_i P_i$, $P \in CL(P)$, $V_0 = \{P\}$, otherwise for each i $P_i \in V_0$, $P_i \in CL(P)$;
2. For all $Q \in CL(P) \setminus \{\varepsilon, false\}$, if Q is rewritten into its normal form $\bigvee_{j=0}^{h}(Q_{ej} \wedge empty) \vee \bigvee_{i=0}^{k}(Q_{ci} \wedge \bigcirc Q'_i)$, then $\varepsilon \in CL(P)$, $(Q, Q_{ej}, \varepsilon) \in EL(P)$ for each j, $1 \leq j \leq h$; $Q'_i \in CL(P)$, $(Q, Q_{ci}, Q'_i) \in EL(P)$ for all i, $1 \leq i \leq k$;

The NFG of formula P is the directed graph $G = (CL(P), EL(P), V_0)$. □

It is important that for any PPTL formula P, the number of $CL(P)$ is finite [4]. An algorithm for constructing NFGs of a formula is given in Algorithm NFG. Although the algorithm is similar as the one given in [4], the proofs of Theorem 2, Lemma 3 and Lemma 4 are totally new, and based on the algorithm.

In an NFG, any root node in V_0 is denoted by a circle with an input edge originating from none nodes, ε node is marked by a small black dot, and each of other nodes by a single circle. Each edge is denoted by a directed arc connecting two nodes. A finite path is a finite alternating sequence of nodes and edges, $\pi = \langle n_0, e_0, n_1, e_1, ..., \varepsilon\rangle$ from a root node to the ε node, while an infinite path is an infinite alternating sequence of nodes and edges, $\pi = \langle n_0, e_0, n_1, e_1 ..., n_i, e_i ..., n_j, e_j, n_i, e_i, ..., n_j, e_j, ...\rangle$ departing from the root node with some nodes, e.g. $n_i, ..., n_j$, occurring for infinitely many times. For convenience, we use $\mathsf{Inf}(\pi)$ to denote the set of nodes which infinitely often occur in the infinite path π. In some circumstances, in a path of NFG of formula Q, a node n_i can be replaced by a formula $Q_i \in CL(Q)$ and an edge e_i can be replaced by a state formula $Q_{ie} \in EL(Q)$.

Intuitively, all models of a formula are implicitly contained in its NFG. For easily expressing the relationship between models of formula Q and paths of NFG G of Q, functions $P2M(\pi, G)$ and $M2P(\sigma, G)$ are formally defined bellow. Given a path $\pi = \langle Q, Q_{0e}, Q_1, Q_{1e}, ...\rangle$ in G, an interval σ_π can be obtained by,

$$\sigma_\pi = P2M(\pi, G) = \begin{cases} \langle s_0, s_1, ..., s_n\rangle, \text{ for } 1 \leq i \leq n \\ \qquad s_i[q] = true \text{ if } q \in Q_{ie}, \text{ and } s_i[q] = false \text{ if } \neg q \in Q_{ie}, \\ \qquad \text{if } \pi = \langle Q, Q_{0e}, Q_1, Q_{1e}, ..., Q_{ne}, \varepsilon\rangle \text{ and } \pi \in G \\ \langle s_0, s_1, ..., (s_i, ..., s_j)^\omega\rangle, \text{ for } 1 \leq k \leq j \\ \qquad s_k[q] = true \text{ if } q \in Q_{ke}, \text{ and } s_k[q] = false \text{ if } \neg q \in Q_{ke}, \\ \qquad \text{if } \pi = \langle Q, Q_{0e}, Q_1, Q_{1e}, ..., (Q_i, Q_{ie}, ..., Q_j, Q_{je})^\omega\rangle \text{ and } \pi \in G \end{cases}$$

Correspondingly, for a model $\sigma = \langle s_0, s_1, ...\rangle \models Q$, a path π_σ w.r.t the NFG G of Q can be obtained by,

$$\pi_\sigma = M2P(\sigma, G) = \begin{cases} \langle Q, Q_{0e}, Q_1, Q_{1e}, ..., Q_{ne}, \varepsilon\rangle, \text{ for } 1 \leq i \leq n \\ \qquad Q_{ie} = \bigwedge_k \dot{q}_k, \dot{q}_k \equiv q_k, \text{ if } s_i[q_k] = true, \\ \qquad \text{and } \dot{q}_k \equiv \neg q_k, \text{ if } s_i[q_k] = false, Q_i \in CL(Q) \\ \qquad \text{if } \sigma = < s_0, s_1, ..., s_n > \\ \langle Q, Q_{0e}, Q_1, Q_{1e}, ..., (Q_i, Q_{ie}, ..., Q_j, Q_{je})^\omega\rangle, \text{ for } 1 \leq h \leq j \\ \qquad Q_{he} = \bigwedge_k \dot{q}_k, \dot{q}_k \equiv q_k, \text{ if } s_h[q_k] = true, \\ \qquad \text{and } \dot{q}_k \equiv \neg q_k, \text{ if } s_h[q_k] = false, Q_h \in CL(Q) \\ \qquad \text{if } \sigma = \langle s_0, s_1, ..., (s_i, ..., s_j)^\omega\rangle \end{cases}$$

Although in the above an interval σ_π can be defined for a given path π of the NFG of formula Q, whether or not $\sigma_\pi \models Q$ needs to be proved (see Lemma 6 and 13). Similarly, given a model σ of formula Q, $\sigma \models Q$, a path π_σ can be constructed, however, whether or not the path can be found in G also needs to be proved (See Lemma 7 and 9).

Theorem 2. Finite paths in the NFG of PPTL formula P precisely characterize finite models of P.
Proof: It is a consequence of Lemma 3 and 4. □

Lemma 3. For a finite path π in the NFG G of P, $\sigma_\pi \models P$. □

Lemma 4. For a finite interval $\sigma \models Q$, π_σ can be found in the NFG G of Q. □

For infinite models, it is much more intricate because of the involvement of chop and projection operators. In fact, not all of infinite paths in the NFG of a formula are infinite models of the corresponding formula. We investigate the case carefully in the following.

A formula R is called a chop formula if $R \equiv P \wedge Q$ and $P \equiv P_1; P_2$, where P_1, P_2 and Q are any PPTL formulas. Further, $P_1; P_2$ is called a chop component of chop formula R. Note that a chop component is also a chop formula but the reverse may be not true. Formally, a chop formula R_c can be defined as follows

$$R_c ::= P; Q \mid R_c \wedge R_c \mid R_c \wedge R$$

where P, Q and R are any PPTL formulas.

For instance, $(\bigcirc p; q)$, $(p; p^*)$, $(\bigcirc p; q) \wedge \bigcirc r$ are chop formulas while $\bigcirc(\bigcirc p; q)$, $\neg(\bigcirc p; q)$ and p^* are not, where p, q, r are atomic propositions. Note that $P; Q$ is a chop formula but $\neg(P; Q)$ is not. This will be formally analyzed late.

For chop construct $P; Q$, an infinite model $\sigma = \langle s_0, s_1, ..., s_k, ...\rangle \models P; Q$ if and only if there exists $i \in N_0$, such that $\sigma^i \models P$ and $\sigma^{(i)} \models Q$. This implies that if infinite model $\sigma = \langle s_0, s_1, ..., s_k, ...\rangle \models P$ and there is no finite prefix $\sigma^i \models P$ ($i \in N_0$), it fails to satisfy $P; Q$. Note that in the weak version of the chop construct, $P; Q$ is still satisfied in the case. For convenience, we say finiteness of P in strong chop construct $P; Q$ (FSC_Property for short) is satisfiable over an infinite model σ, denoted by predicate $fsc(\sigma, P; Q) = true$, if there exists $i \in N_0$ such that $\sigma^i \models P$ and $\sigma^{(i)} \models Q$. Actually, an infinite path of the NFG of $P; Q$ presents an infinite model of either $P; Q$ or P. So, FSC_Property needs to be considered if a node (formula) is a chop formula.

Correspondingly, in NFGs, when constructing NFG of $P; Q$, initially, $P; Q$ is transformed into its normal form,

$$\begin{aligned} P; Q &\equiv (\bigvee_{j=0}^{n_0} P_{ej} \wedge empty \vee \bigvee_{i=0}^{n_1} (P_{ci} \wedge \bigcirc P'_i)); Q \\ &\equiv \bigvee_{j=0}^{n_0} (\underline{P_{ej} \wedge empty; Q}) \vee \bigvee_{i=0}^{n_1} P_{ci} \wedge \bigcirc(P'_i; Q) \\ &\equiv \bigvee_{j=0}^{n_0} (P_{ej} \wedge Q) \vee \bigvee_{i=0}^{n_1} P_{ci} \wedge \bigcirc(P'_i; Q) \end{aligned}$$

Subsequently, the new generated formula $P'_i; Q$ needs to be repeatedly transformed into its normal form in the same way. Whenever a final state of P is reached, i.e. $P_e \wedge empty; Q$ is encountered, where P_e is a state formula, FSC_Property of $P; Q$ is satisfied over the path π departing from the root node.

Formally, let $\pi = \langle R, R_{0e}, R_1, R_{1e}, ...\rangle$ be an infinite path in the NFG of formula R, and $\pi^{(k)} = \langle R_k, R_{ke}, ...\rangle$ be the kth suffix of π. Whether or not FSC_Property of R is satisfiable over π, denoted by predicate $FSC(\pi, R)$, can be defined based on $fsc(\pi, R)$ as follows.

Definition 4 . $fsc(\pi, R) = true$ if

1. R is not a chop formula, or
2. $R \equiv P; Q$ and there exists $i \in N_0$ such that $R_i \equiv P_{ie} \wedge empty; Q$, or
3. $R \equiv P \wedge Q$ and $fsc(\pi, P) = true$ and $fsc(\pi, Q) = true$, or
4. $R \equiv P \vee Q$ and $fsc(\pi, P) = true$ or $fsc(\pi, Q) = true$

Finally, $FSC(\pi, R) = true$ if $fsc(\pi^{(k)}, R_k) = true$ for all $k \in N_0$. □

In particular, notice that $fsc(\pi, \neg R) = true$ if $R \equiv P; Q$, since in $\neg(P; Q)$, FSC_Property of $P; Q$ needs not to be considered.

Lemma 5. For an infinite model, $\sigma \models \neg(P; Q)$ iff $\forall k \in N_\omega \wedge (\sigma^k \models \neg P \vee \sigma^{(k)} \models \neg Q)$. □

For projection construct, $(P_1, ..., P_m)\ prj\ Q$, it is eventually treated as a chop formula when constructing NFGs according to the following transforming rule. Suppose

$$\begin{aligned} P_1 &\equiv P_{1e} \wedge empty \vee \bigvee_{i=1}^{r}(P_{1i} \wedge \bigcirc P'_{1i}) \\ P_2 &\equiv P_{2e} \wedge empty \vee \bigvee_{l=1}^{s}(P_{2l} \wedge \bigcirc P'_{2l}) \\ Q &\equiv Q_e \wedge empty \vee \bigvee_{j=1}^{k}(Q_j \wedge \bigcirc Q'_j) \end{aligned}$$

then,

$$\begin{aligned} (P_1, P_2)\ \text{prj}\ Q \equiv\ & Q_e \wedge P_{1e} \wedge P_{2e} \wedge empty \\ & \vee \bigvee_{l=1}^{s}(Q_e \wedge P_{1e} \wedge P_{2l} \wedge \bigcirc P'_{2l}) \\ & \vee \bigvee_{j=1}^{k}(P_{1e} \wedge P_{2e} \wedge Q_j \wedge \bigcirc Q'_j) \\ & \vee \bigvee_{j=1}^{k} \bigvee_{l}^{s}(P_{1e} \wedge Q_j \wedge P_{2l} \wedge \bigcirc(P'_{2l}\ ;\ Q'_j)) \\ & \vee \bigvee_{i=1}^{r}(Q_e \wedge P_{1i} \wedge \bigcirc(P'_{1i}\ ;\ P_2)) \\ & \vee \bigvee_{i=1}^{r} \bigvee_{j}^{k}(Q_j \wedge P_{1i} \wedge \bigcirc(P'_{1i}\ ;\ (P_2\ \text{prj}\ Q'_j))) \end{aligned}$$

The above explanation shows that the fitness property for projection constructs and $\neg(P; Q)$ needs not to be considered, and only for chop formulas requires to be treated in a special way. To do so, we need the fixed point theory and Scott's fixed point induction.

Fixed point theory: Every monotonic function F over a complete lattice $\langle A, \subseteq \rangle$ has a unique least fixed point $\bigcup_i F^i(\bot)$ and a unique greatest fixed point $\bigcap_i F^i(\top)$ [14]. □

Scott's fixed-point induction: Suppose D is a complete lattice with bottom $\bot$, $F : D \rightarrow D$ is a continuous function, and P is an inclusion subset of D. If $\bot \in P$ and $\forall x \in D.x \in P \Rightarrow F(x) \in P$, then $fix(F) \in P$ [14]. □

In the following, Lemma 6, 7, 8, 10 and Theorem 11 represent the relationship between infinite models and paths in the NFG of a formula.

Lemma 6. For an infinite model σ, $\sigma \models Q$, a π_σ with $FSC(\pi_\sigma, Q) = true$ can be found in the NFG G of Q. □

Lemma 7. In the NFG of R, for an infinite path π, if there exist no chop formulas over π, $\sigma_\pi \models R$. □

In the following, we further discuss the general case of path π for a given formula R. The final conclusion is presented in Theorem 11.

Lemma 8. In the NFG G of formula R, for an infinite path $\pi = \langle R, R_{e0}, R_1, R_{e1}, R_2, ...\rangle$, if $\sigma_\pi^{(i)} \models R_i$, then $\sigma_\pi^{(i-1)} \models R_{i-1}$. □

Corollary 9. In the NFG of formula R, for an infinite path $\pi = \langle R, R_{e0}, R_1, R_{e1}, R_2, ...\rangle$, $\sigma_\pi \models R$ if there exists $i \in N_0$ such that $\sigma_\pi^{(i)} \models R_i$. □

Lemma 10. If π is an infinite path in the NFG G of formula R with $FSC(\pi, R) = true$, and $\sigma_\pi = P2M(\pi, G)$, then $\sigma_\pi \models R$. □

Theorem 11. In the NFG of PPTL formula R, an infinite path π with $FSC(\pi, R) = true$ precisely characterizes infinite models of R.

Proof: The theorem is a direct consequence of Lemma 6 and 10. □

4 Decision Procedure Based on LNFG

To explicitly display whether or not the FSC_Property of a chop formula is satisfied, extra propositions $\mathsf{l_k}$, $k \in N_0$ and $k > 0$, are introduced. Let $Prop_l = \{\mathsf{l_1}, \mathsf{l_2}, ...\}$ be the set of extra propositions. $Prop \cap Prop_l = \emptyset$. Note that these extra propositions are merely employed to mark nodes and are not allowed to appear in a PPTL formula. When constructing NFGs by normal forms, for any chop formula $P; Q$, we equivalently rewrite it as $P \wedge fin(\mathsf{l_k}); Q$. Formally, we have,

$$\begin{aligned}
& P \wedge fin(\mathsf{l_k}); Q \\
\equiv\ & (\textstyle\bigvee_{j=0}^{n_0} P_{ej} \wedge empty \vee \bigvee_{i=0}^{n_1}(P_{ci} \wedge \bigcirc P'_i)) \wedge (\mathsf{l_k} \wedge empty \vee \bigcirc fin(\mathsf{l_k})); Q \\
\equiv\ & (\textstyle\bigvee_{j=0}^{n_0} P_{ej} \wedge \mathsf{l_k} \wedge empty \vee \bigvee_{i=0}^{n_1}(P_{ci} \wedge \bigcirc(P'_i \wedge fin(\mathsf{l_k})))); Q \\
\equiv\ & \textstyle\bigvee_{j=0}^{n_0}(\underline{P_{ej} \wedge \mathsf{l_k} \wedge empty; Q}) \vee \bigvee_{i=0}^{n_1}(P_{ci} \wedge \bigcirc(P'_i \wedge fin(\mathsf{l_k}); Q)) \\
\equiv\ & \textstyle\bigvee_{j=0}^{n_0}(\underline{P_{ej} \wedge \mathsf{l_k} \wedge Q}) \vee \bigvee_{i=0}^{n_1}(P_{ci} \wedge \bigcirc(P'_i \wedge fin(\mathsf{l_k}); Q))
\end{aligned}$$

Thus, by using $fin(\mathsf{l_k})$, FSC_Property of $P; Q$ is satisfied if there exists an edge where $\mathsf{l_k}$ holds. And $fin(\mathsf{l_k})$ occurring in a node $P \wedge fin(\mathsf{l_k}); Q$ means that FSC_Property of $P; Q$ has not been satisfied at this node. For convenience, for a node in the form of $P \wedge fin(\mathsf{l_k}); Q$ or $\bigwedge_{i=1}^{n} R_i$ with some $R_i \equiv P_i \wedge fin(\mathsf{l_k}); Q_i$, we add an extra label $\tilde{\mathsf{l_k}}$ in this node to mean that the finiteness of some chop formula has not been satisfied at this node.

Accordingly, Labeled Normal Form Graph (LNFG) is defined based on NFG with the usage of $\mathsf{l_k}$ propositions.

Definition 5 (Labeled Normal Form Graph, LNFG). For a PPTL formula P, its LNFG is a tuple $G = (CL(P), EL(P), V_0, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$, where $CL(P)$, $EL(P)$ and V_0 are identical to the ones in NFG, while each $\mathbb{L}_k \subseteq CL(P)$, $1 \le k \le m$, is the set of nodes with $\tilde{\mathsf{l_k}}$ labels. □

Algorithm LNFG: Constructing LNFG of a PPTL formula.

Function LNFG(P)
/* precondition: P is a PPTL formula*/
/* postcondition: LNFG(P) computes LNFG of P, $G = (CL(P), EL(P), V_0, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$*/
begin function
  case
    P is $\bigvee_i P_i$: $CL(P) = \{P_i | P_i$ appears in $\bigvee_i P_i\}$; $Mark[P_i] = 0$ for each i;
        $V_0 = \{P_i | P_i$ appears in $\bigvee_i P_i\}$;
    P is not $\bigvee_i P_i$: $CL(P) = \{P\}$; $Mark[P] = 0$; $V_0 = \{P\}$;
  end case
  $EL(P) = \emptyset$; AddE = AddN = 0; $k = 0$; $\mathbb{L} = \emptyset$;
  while there exists $R \in CL(P) \setminus \{\varepsilon, false\}$, and $mark[R]$ ==0
    if $R \equiv P; Q$ or $R \equiv \bigwedge_{i=1}^{n} R_i$ and $\exists R_i \equiv P_i; Q_i$ and no $fin(k)$ has been added in the chop construct
      k=k+1; Rewrite R as $P \wedge fin(\mathsf{l_k}); Q$ or $\bigwedge_{i=1}^{n} R_i$ with some R_i being $P_i \wedge fin(\mathsf{l_k}); Q_i$;
        $\mathbb{L}_k = \{R\}$; $\mathbb{L} = \mathbb{L} \cup \{\mathbb{L}_k\}$;
      if there exist $P \wedge fin(\mathsf{l_s}); Q$ or $\bigwedge_{i=1}^{n} R'_i$ with some $R'_i \equiv P \wedge fin(\mathsf{l_s}); Q$, and $P \wedge fin(\mathsf{l_x}); Q$ or $\bigwedge_{i=1}^{n} R'_i$ with some $R'_i \equiv P \wedge fin(\mathsf{l_x}); Q$, $1 \leq l < x < k$
        delete the node $P \wedge fin(\mathsf{l_k}); Q$ or $\bigwedge_{i=1}^{n} R'_i$ with some $R'_i \equiv P \wedge fin(\mathsf{l_s}); Q$;
        re-adding the edges to $P \wedge fin(\mathsf{l_s}); Q$ or $\bigwedge_{i=1}^{n} R'_i$; continue;
    Q =NF(R); $mark[R]$ =1;   /*marking R is decomposed*/
    case
      Q is $\bigvee_{j=1}^{h} Q_{ej} \wedge empty$: AddE=1;   /*first part of NF needs added*/
      Q is $\bigvee_{i=1}^{k} Q_i \wedge \bigcirc Q'_i$: AddN=1;   /*second part of NF needs added*/
      Q is $\bigvee_{j=1}^{h} Q_{ej} \wedge empty \vee \bigvee_{i=1}^{k} Q_i \wedge \bigcirc Q'_i$: AddE=AddN=1;   /*both parts need added*/
    end case
    if AddE == 1 **then**   /*add first part of NF*/
      $CL(P) = CL(P) \cup \{\varepsilon\}$; $EL(P) = EL(P) \cup \bigcup_{j=1}^{h}\{(R, Q_{ej}, \varepsilon)\}$; AddE=0;
    if AddN == 1 **then for** $i = 1$ **to** k,   /*add second part of NF*/
      if $Q'_i \notin CL(P)$,$CL(P) = CL(P) \cup \bigcup_{i=1}^{k}\{Q'_i\}$;
        if Q'_i is not $false$, $mark[Q'_i]$=0; **else** $mark[Q'_i]$=1;
      $EL(P) = EL(P) \cup \bigcup_{i=1}^{k}\{(R, Q_i, Q'_i)\}$; AddN=0;
  end while
  return G;
End function

Algorithm LNFG based on Algorithm NFG is given by further rewriting the chop component $P; Q$ as $P \wedge fin(\mathsf{l_k}); Q$ for some $k \in N_0$ whenever a new chop formula is encountered. The algorithm uses *mark*[] to indicate whether or not a node needs to be decomposed. If $mark[P] = 0$ (unmarked), P needs further to be decomposed, otherwise if $mark[P] = 1$ (marked), P has been decomposed or needs not to be done. Function NF is used to transform a formula into its normal form. Further, in the algorithm, two global boolean variables AddE and AddN are employed to indicate whether or not terminating part and non-terminating part of the normal form are encountered respectively.

Lemma 12. For an infinite path π in the LNFG, $G = (CL(R), EL(R), V_0, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$, of PPTL formula R, $FSC(\pi, Q) = true$ iff $\mathsf{Inf}(\pi) \not\subseteq \mathbb{L}_i$ for any $1 \leq i \leq m$. □

Theorem 13. In the LNFG of a formula P, finite paths precisely characterize finite models of P; infinite paths with $\mathsf{Inf}(\pi) \not\subseteq \mathbb{L}_i$, for all $1 \leq i \leq m$ precisely characterize infinite models of P.

Proof: For finite case, it has been proved in Theorem 2. For infinite case, it is a direct consequence of Theorem 11 and Lemma 12. □

Consequently, a decision procedure for checking the satisfiability of a PPTL formula P can be constructed based on the LNFG of P. In the following, a sketch of the procedure, Algorithm CHECK in pseudo code, is demonstrated.

In order to use our approach, we have developed a tool in C++. With this tool, all the algorithms introduced above have been implemented. For any PPTL formula, the tool can automatically transform it to its normal form, and LNFG, then check whether the formula is satisfiable or not.

Algorithm CHECK: Checking whether or not P is satisfiable.

Function CHECK(P)
/* precondition: P is a PPTL formula*/
/* postcondition: CHECK(P) checks whether formula P is satisfiable or not.*/
begin function
    G =LNFG(P);
    if there exists ε node in $CL(P)$,
    return P is satisfiable with finite models;
    if there exists infinite path π with $\mathsf{Inf}(\pi) \not\subseteq \mathbb{L}_i$, for all $1 \leq i \leq m$
    return P is satisfiable with infinite models;
    else return unsatisfiable;
end function

Example 1. Checking the satisfiability of formula $P \equiv (p \wedge \Box \bigcirc p; \bigcirc \Box q) \wedge (\Box r; \bigcirc \Box q)$.

By Algorithm LNFG, LNFG $G = (CL(P), EL(P), V_0, \ \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$ of formula $(p \wedge \Box \bigcirc p; \bigcirc \Box q) \wedge (\Box r; \bigcirc \Box q)$ is constructed as depicted in Fig.3, where $CL(P) = \{n_0, n_1\}$, $EL(P) = \{(n_0, p \wedge r, n_0), (n_0, p \wedge r, n_1), (n_1, p, n_1)\}$, $V_0 = \{n_0\}$, $\mathbb{L} = \{\mathbb{L}_1, \mathbb{L}_2\}$, $\mathbb{L}_1 = \{n_0, n_1\}$ and $\mathbb{L}_2 = \{n_0\}$. The formula is unsatisfiable since there exists no node without $\tilde{\mathsf{L}}_1$ label which is reachable from n_0 and n_1. □

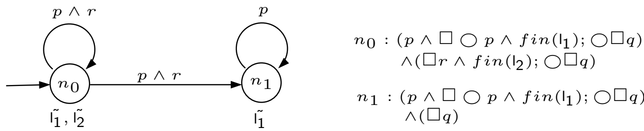


Fig. 3. LNFG of $(p \wedge \Box \bigcirc p; \bigcirc \Box q) \wedge (\Box r; \bigcirc \Box q)$

Example 2. Re-checking the formula given in the introduction.

The example presented in the introduction, which cannot be handled by the old algorithm can now properly be treated. The new LNFG of formula $q \wedge (\bigcirc empty; q) \wedge \Box(p \wedge \bigcirc\bigcirc empty; q)$ is illustrated in Fig.4. The formula is satisfiable since for the only infinite path $Inf(\pi) \not\subseteq \mathbb{L}_i$, $i = 1, 2$. □

$n_0 : q \wedge (\bigcirc empty \wedge fin(l_1); q) \wedge \Box(p \wedge \bigcirc \bigcirc empty; q)$

$n_1 : q \wedge (\bigcirc empty \wedge fin(l_2); q) \wedge \Box(p \wedge \bigcirc \bigcirc empty; q)$

Fig. 4. LNFG of $q \wedge (\bigcirc empty; q) \wedge \Box(p \wedge \bigcirc \bigcirc empty; q)$

Example 3. Checking the satisfiability of formula $(p \wedge skip; q \wedge skip)^+ \wedge \Box more; r$.

The formula is obviously unsatisfiable since $(p \wedge skip; q \wedge skip)^+ \wedge \Box more$ cannot be satisfied with finite models. However, by the old algorithm, the LNFG of $(p \wedge skip; q \wedge skip)^+ \wedge \Box more$ can be constructed as illustrated in Fig.5 (1) which shows the formula is satisfiable. With the new algorithm, the LNFG is depicted in Fig.5 (2) which indicates the formula is unsatisfiable. □

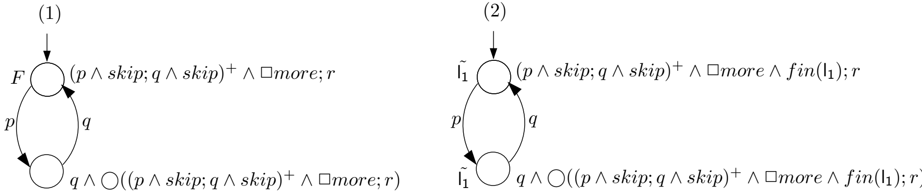


Fig. 5. LNFG of $(p \wedge skip; q \wedge skip)^+ \wedge \Box more; r$

5 Model Checking PPTL

Based on the new decision procedure, the PPTL model checker based on SPIN has also been improved. To this end, a new transformation from LNFGs to Büchi automata is presented. We first transform an LNFG to a Generalized Büchi Automata (GBA) [16]. Factually, an LNFG contains all the information of the corresponding GBA. The set of nodes is in fact the set of locations in the corresponding GBA; each edge (v_i, Q_e, v_j) forms a transition; there exists only one initial location, the root node; the set of accepting locations consists of ε node and the nodes which can appear in infinite paths for infinitely many times. Given an LNFG $G = (CL(P), EL(P), V_0, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$ of formula P, an GBA, $B = (Q, I, \delta, F = \{F_1, ..., F_m\})$, can be constructed as follows.

- Sets of the locations Q and the initial locations I: $Q = V$, and $I = \{v_0\}$.
- Transition δ: Let $\dot{q}_k$ be an atomic proposition or its negation, and we define a function $atom(\bigwedge_{k=1}^{m_0} \dot{q}_k)$ for picking up atomic propositions or their negations appearing in $\bigwedge_{k=1}^{m_0} \dot{q}_k$ as follows,

$$atom(true) = true$$
$$atom(\dot{q}_k) = \begin{cases} \{q_k\}, & \text{if } \dot{q}_k \equiv q_k \ 1 \leq k \leq l \\ \{\neg q_k\}, & \text{otherwise} \end{cases}$$
$$atom(\textstyle\bigwedge_{k=1}^{m_0} \dot{q}_k) = atom(\dot{q}_1) \cup atom(\textstyle\bigwedge_{k=2}^{m_0} \dot{q}_k)$$

For each $e_i = (v_i, Q_e, v_{i+1}) \in E$, there exists $v_{i+1} \in \delta(v_i, atom(Q_e))$. For node ε, $\delta(\varepsilon, \epsilon) = \{\varepsilon\}$.

– Accepting set $F = \{F_1, ..., F_m\}$: It has been proved that infinite paths with $\mathsf{Inf}(\pi) \not\subseteq \mathbb{L}_i$ for all $1 \leq i \leq m$ precisely characterize infinite models of P. This can be equivalently expressed by "infinite paths with $\mathsf{Inf}(\pi) \cap \overline{\mathbb{L}_i} \neq \emptyset$ for all $1 \leq i \leq m$ precisely characterize infinite models of P", where $\overline{\mathbb{L}_i}$ denotes $CL(P) \setminus \mathbb{L}_i$. So, $F_i = \overline{\mathbb{L}_i}$ for each i. In addition, by employing the stutter extension rule, $\{\varepsilon\}$ is also an accepting set.

Formally, algorithm Lnfg-Gba in pseudo code is given for transforming an LNFG to a GBA.

Algorithm Lnfg-Gba: Transforming an LNFG to a GBA.

Function Check(P)
/* precondition: $G = (CL(P), EL(P), V_0,\ \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$ is the LNFG of PPTL formula P*/
/* postcondition: Lnfg-Sba(G) computes an GBA $B = (Q, I, \delta, F = \{F_1, ..., F_n\})$ from G*/
begin function
  $Q = \emptyset$; $F = \{F_1, ..., F_m\}$; $F_i = \emptyset$, $1 \leq i \leq m$; $I = \emptyset$;
  for each node $v_i \in V$, add a state q_i to Q, $Q = Q \cup \{q_i\}$; **end for**
  for each node $v_i \in \overline{\mathbb{L}_i}$, add a state q_i to F_i, $F_i = F_i \cup \{q_i\}$;**end for**
  if v_i is ε, $F = F \cup \{q_i\}$; $\delta(q_i, \epsilon) = \{q_i\}$;
  if $q_0 \in V_0$, $I = I \cup \{q_0\}$;
  for each edge $e = (v_i, P_e, v_j) \in E$, $q_j \in \delta(q_i, atom(P_e))$;**end for**
 return $B = (Q, \Sigma, I, \delta, F)$
end function

Consequently, the algorithm presented in [16] for transforming a GBA to a Büchi automaton can be applied to further transform the GBAs to Bü chi automata.

By transforming PPTL formulas to Büchi automata which precisely characterize the models of the corresponding formulas, an automaton based model checking algorithm for PPTL can be formalized. The algorithm can further be implemented within the model checker SPIN. With our model checking algorithm for PPTL, the system to be verified is modeled as a B*ü*chi automaton A_s, while the property is specified by a PPTL formula P. To check whether the system satisfies P or not, $\neg P$ is transformed into an LNFG, and further a B*ü*chi automaton A_p. The system can be verified by computing the product automaton of A_s and A_p, and then checking whether the words accepted by the product automaton is empty or not as shown in Fig. 6. If the words accepted by the product automaton is empty, the system can satisfy the property otherwise the system cannot satisfy the property, and a counter-example can be found.

With SPIN, a translator from PLTL to Never Claim, and a B*ü*chi automaton in terms of PROMELA, are realized to automatically transform a formula in linear temporal logic to Never Claim. To implement model checking PPTL in SPIN, we also provide a translator from PPTL formulas to Never Claims as shown in Fig. 7. We have realized the translator in C++ according to the algorithms presented in this paper. Further, the translator has successfully been integrated within the original SPIN.

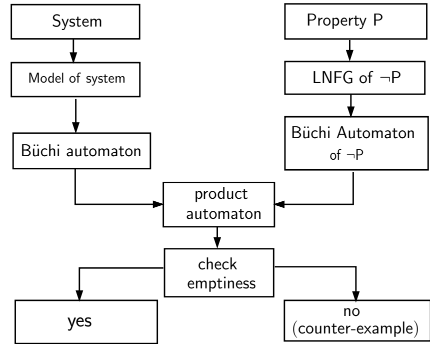


Fig. 6. Model checking PPTL

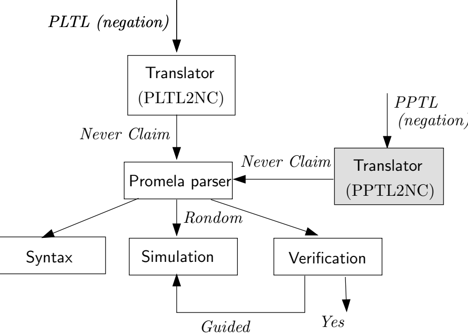


Fig. 7. The model checker SPIN with PPTL

6 Conclusion

An improved decision procedure for checking satisfiability of PPTL formulas is given in this paper. The algorithm has been realized and a model checker based on it has been developed recently. With our experience, the decision procedure and the model checker are useful in practice since full regular properties can be described by PPTL. In the future, we will further investigate the partial order model checking, symbolic model checking, bounded model checking and compositional model checking with PPTL. Furthermore, some supporting tools will also be developed to support our model checking algorithms.

References

1. Kripke, S.A.: Semantical analysis of modal logic I: normal propositional calculi. Z. Math. Logik Grund. Math. 9, 67–96 (1963)
2. Moszkowski, B.: Reasoning about digital circuits, Ph.D Thesis, Department of Computer Science, Stanford University, TRSTAN-CS-83-970 (1983)

3. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne (May 1996)
4. Duan, Z., Tian, C., Zhang, L.: A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. Acta Informatica 45(1), 43–78 (2008)
5. Wang, H., Xu, Q.: Temporal logics over infinite intervals. Technical Report 158, UNU/IIST, Macau (1999)
6. Moszkowski, B.C.: A Complete Axiomatization of Interval Temporal Logic with Infinite Time. In: 15th Annual IEEE Symposium on Logic in Computer Science. LICS, p. 241 (2000)
7. Chaochen, Z., Hoare, C.A.R., Ravn, A.P.: A calculus of duration. Information Processing Letters 40(5), 269–275 (1991)
8. Bowman, H., Thompson, S.: A decision procedure and complete axiomatization of interval temporal logic with projection. Journal of logic and Computation 13(2), 195–239 (2003)
9. Dutertre, B.: Complete proof systems for first order interval temporal logic. In: Proceedings of LICS 1995, pp. 36–43 (1995)
10. Clark, M., Gremberg, O., Peled, A.: Model Checking. The MIT Press, Cambridge (2000)
11. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Found. of Comp. Sci., pp. 46–57 (1977)
12. Tian, C., Duan, Z.: Propositional Projection Temporal Logic, Büchi Automata and ω-Expressions. In: Agrawal, M., Du, D.-Z., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
13. Tian, C., Duan, Z.: Complexity of Propositional Projection Temporal Logic with Star. Mathematical Structure in Computer Science 19(1), 73–100 (2009)
14. Winskel, G.: The Formal Semantics of Programming Languages. In: Foundations of Computing. The MIT Press, Cambridge
15. Vardi, M.Y.: The Büchi Complementation Saga. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 12–22. Springer, Heidelberg (2007)
16. Katoen, J.-P.: Concepts, Algorithms, and Tools for Model Checking. Lecture Notes of the Course Mechanised Validation of Parrel Systems (1999)
17. Tian, C., Duan, Z.: Model Checking Propositional Projection Temporal Logic Based on SPIN. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 246–265. Springer, Heidelberg (2007)

A Semantic Model for Service Composition with Coordination Time Delays

Natallia Kokash, Behnaz Changizi, and Farhad Arbab[⋆,⋆⋆]

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Natallia.Kokash@cwi.nl

Abstract. The correct behavior of a service composition depends on the appropriate coordination of its services. According to the idea of channel-based coordination, services exchange messages though channels without any knowledge about each other. The Reo coordination language aims at building connectors out of basic channels to implement arbitrarily complex interaction protocols. The activity within a Reo connector consists of two types of communication, each of which incurs a delay: internal coordination and data transfer. Semantic models have been proposed for Reo that articulate data transfer delays, but none of them explicitly considers coordination delays. More importantly, these models implicitly assume that (1) internal coordination and data transfer activities take place in two separate phases, and (2) data transfer delays do not affect the coordination phase. This assumptions prevent maximal concurrency in data exchange and distort the evaluation of end-to-end delays in service composition models. In this paper, we introduce a novel compositional automata-based semantic model for Reo that explicitly represents both internal coordination and data transfer aspects in channel-based connectors. Furthermore, we map the proposed model to the process algebra `mCRL2`, which allows us to generate state spaces for connectors with time delays and analyze them automatically.

1 Introduction

Provisioning of end-to-end client-perceived Quality of Service (QoS) in concurrent systems is a well-renown problem that has been attracting attention of researchers and software engineers over the past few decades. The problem acquired even more attention with the advent of service-oriented computing where systems are composed out of loosely-coupled services of different vendors to realize complex value-added business processes. The quality of what a service-based system offers is derived from the quality of its constituent parts, the quality of the so-called "glue code" that coordinates the execution of the individual services, and the characteristics of the underlying infrastructure such as, e.g., the physical location of servers and the characteristics of the communication networks connecting them.

⋆ Corresponding author.

⋆⋆ Supported by IST COMPAS FP7-ICT-2007-1 project, contract number 215175.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 106–121, 2010.
© Springer-Verlag Berlin Heidelberg 2010

One of the ways to coordinate autonomous services is to use connectors. Reo [1] is a model for coordination of software components or services wherein complex connectors are constructed out of simple primitives called channels. By composing basic channels, arbitrarily complex interaction protocols can be implemented. A distinctive characteristic of Reo is the propagation of synchrony: with the help of connectors composed of synchronous channels, one can define transactional protocols where all participating services should be ready to provide or consume messages simultaneously. This facility is very useful as it enables models that are both concise and compositional, but it also makes the problem of describing the operational semantics of Reo a non-trivial task. The most basic semantic model that currently exists for Reo relies on constraint automata [2]. In this model, states represent configurations of data stored in the buffers of Reo networks, while transition labels are composed of (i) sets of channel ends where dataflow is observed simultaneously, and (ii) data constraints necessary to trigger such transitions. Constraint automata represent a theoretical basis for most of the available validation and verification tools for Reo, which are integrated in a framework known as the Eclipse Coordination Tools (ECT)[1].

Several extensions for Reo and its initial semantics have been proposed to capture the notions of timed, context-sensitive, probabilistic and stochastic behavior. However, none of these models accounts for the possible delays that the channels need to transfer data. The existing approaches that aim at extending Reo with QoS information [3,4] assume that delays in channels do not affect the operational semantics of Reo connectors. Nevertheless, as we argue in this paper, this assumption can limit the degree of concurrency in the presence of transactions with different durations and lead to imprecise estimations of end-to-end communication delays in service compositions.

To fix this problem, we introduce a more expressive semantic model for Reo, called *action constraint automata*, which distinguished several actions performed internally by each channel to manifest its behavior. By observing the start and the end of a multiparty communication as well as the start and the end of actual dataflow in each channel, we include more information into the model describing the behavior of a circuit. This approach eventually helps us to compute the total delay in a circuit given the delays for each individual channel. In this paper, we introduce the action constraint automata model, illustrate its application to dataflow modeling in Reo, and discuss the tool support achieved by mapping action constraint automata into the process algebra `mCRL2` as well as the integration of the `mCRL2` toolset within ECT.

The remainder of this paper is organized as follows. In Section 2, we explain the basics of Reo. In Section 3, we give examples that motivate our work. In Section 4, we introduce the action constraint automata-based semantic model for Reo. In Section 5, we use this new automata to give semantics to our motivating examples. In Section 6, we discuss the translation of this model to the process algebra `mCRL2`, which enables the application of the `mCRL2` toolset for

[1] http://reo.project.cwi.nl/

the verification of Reo circuits. Finally, in Section 7, we conclude the paper and outline our future work.

2 Background

Reo is a coordination language in which components and services are coordinated exogenously by channel-based connectors [1]. Connectors are essentially graphs where the edges are user-defined communication channels and the nodes implement a fixed routing policy. Channels in Reo are entities that have exactly two ends, also referred to as ports, which can be either source or sink ends. Source ends accept data into, and sink ends dispense data out of their channel. Although channels can be defined by users, a set of basic Reo channels with predefined behavior suffices to implement rather complex coordination protocols. Among these channels are (i) the Sync channel, which is a directed channel that accepts a data item through its source end if it can instantly dispense it through its sink end; (ii) the LossySync channel, which always accepts a data item through its source end and tries to instantly dispense it through the sink end. If this is not possible, the data item is lost; (iii) the SyncDrain channel, which is a channel with two source ends that accept data simultaneously and loses them subsequently; (iv) the AsyncDrain channel, which accepts data items only through one of its two source channel ends at a moment in time and loses it; and (v) the FIFO channel, which is an asynchronous channel with a buffer of capacity one. Additionally, there are channels for data manipulation. For instance, the Filter channel always accepts a data item at its source end and synchronously passes or loses it depending on whether or not the data item matches a certain predefined pattern or data constraint. Finally, the Transform channel applies a user-defined function to the data item received at its source end and synchronously yields the result at its sink end.

Channels can be joined together using nodes. A node can be a source, a sink or a mixed node, depending on whether all of its coinciding channel ends are source ends, sink ends or a combination of both. Source and sink nodes together form the boundary nodes of a connector, allowing interaction with its environment. Source nodes act as synchronous replicators, and sink nodes as non-deterministic mergers. A mixed node combines these two behaviors by atomically consuming a data item from one of its sink ends at the time and replicating it to all of its source ends.

The basic set of Reo channels can be extended to enable modeling of specific features of service communication. Apart from functional aspects, channels can differ at the level of their non-functional characteristics. In quantitative Reo [3], channels are characterized by a set of associated QoS parameters such as data transfer delays or cost.

In this paper, we consider Reo channels in presence of internal coordination and data transfer delays. Roughly, by internal coordination delay we mean the time that it takes a channel to decide whether or not it can accept to satisfy the I/O request at its ends. By data transfer delay we mean the time needed

to deliver a data item accepted by the source end of a channel to its sink end for the Sync channels, from the channel source end to its buffer and from the buffer to its sink end for the FIFO channels, or accept and destroy a data item for SyncDrain, AsyncDrain and LossySync channels.

An informal description of Reo given above is rather incomplete and ambiguous. The semantics of any Reo connector can be understood only in terms of a specific semantic model and its appropriate translation into that model.

The most basic model expressing the semantics of Reo formally is constraint automata [2]. Transitions in a constraint automaton are labeled with sets of ports that fire synchronously, as well as with data constraints on these ports. The constraint automata-based semantics for Reo is compositional, meaning that the behavior of a complex Reo circuit can be obtained from the semantics of its constituent parts using the product operator. Furthermore, the hiding operator can be used to abstract from unnecessary details such as dataflow on the internal ports of a connector.

Definition 1 (Constraint Automaton (CA)). *A constraint automaton* $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ *consists of a set of states* S*, a set of port names* $\mathcal{N}$*, a transition relation* $\rightarrow \subseteq S \times 2^{\mathcal{N}} \times DC \times S$*, where* DC *is the set of data constraints over a finite data domain* $Data$*, and an initial state* $s_0 \in S$*.*

We write $q \xrightarrow{N,g} p$ instead of $(q, N, g, p) \in \rightarrow$. Table 1 shows our graphical notation for the basic Reo channels and nodes together with their constraint automata semantics. The behavior of any Reo circuit can be obtained by computing the product of these automata which can be defined using the notion of a port synchronization function [5].

Definition 2. *Let* $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \rightarrow_1, s_0^1)$*,* $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \rightarrow_2, s_0^2)$ *be two constraint automata with disjoint sets of port names* $\mathcal{N}_1$ *and* $\mathcal{N}_2$*, respectively. A* port synchronization function $\gamma \colon \mathcal{N} \rightarrow \mathcal{N}_1 \times \mathcal{N}_2$ *is defined as* $\gamma(n) = (\gamma_1(n), \gamma_2(n))$ *through the pair of injective functions* $\gamma_1 \colon \mathcal{N} \rightarrow \mathcal{N}_1$ *and* $\gamma_2 \colon \mathcal{N} \rightarrow \mathcal{N}_2$ *that map port names from a new set* $\mathcal{N}$ *into port names from the sets* $\mathcal{N}_1$ *and* $\mathcal{N}_2$*.*

In the above definition, "new set" means $\mathcal{N} \cap (\mathcal{N}_1 \cup \mathcal{N}_2) = \emptyset$. Observe that the disjointness of $\mathcal{N}_1$ and $\mathcal{N}_2$ and the injectivity of the total functions γ_1 and γ_2 confine the cardinality of $\mathcal{N}$ such that $0 \leq |\mathcal{N}| \leq min(|\mathcal{N}_1|, |\mathcal{N}_2|)$. The exact definition of γ_1 or γ_2, then, uniquely defines $\mathcal{N}$. Intuitively, $\gamma(n) = (x, y)$ represents a renaming of $x \in \mathcal{N}_1$ and $y \in \mathcal{N}_2$ to the same common element $n \in \mathcal{N}$. In the context of the port synchronization function γ, we write $\mathcal{N}_1'$ for $\mathcal{N}_1 \backslash \gamma_1[\mathcal{N}]$ and $\mathcal{N}_2'$ for $\mathcal{N}_2 \backslash \gamma_2[\mathcal{N}]$. If, for subsets $N_1 \subseteq \mathcal{N}_1$, $N_2 \subseteq \mathcal{N}_2$, it holds that $\gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$ we write

$$N_1 \,|_\gamma\, N_2 = (N_1 \cap \mathcal{N}_1') \cup \gamma_1^{-1}[N_1] \cup (N_2 \cap \mathcal{N}_2') \,.$$

This means that $N_1 \,|_\gamma\, N_2$ is the union $N_1 \cup N_2$ with the parts of N_1 and N_2 that are identified via γ_1 and γ_2 replaced by the shared names $\gamma_1^{-1}[N_1] = \gamma_2^{-1}[N_2]$.

Also, for a constraint g, we write $\gamma(g)$ for the formula obtained by replacing the port names in $\gamma_1[\mathcal{N}] \subseteq \mathcal{N}_1$ and $\gamma_2[\mathcal{N}] \subseteq \mathcal{N}_2$ by the corresponding name in $\mathcal{N}$.

Table 1. Graphical notation and semantics for channels and nodes

Primitive	Notation	Constraint automaton
Sync	$A \longrightarrow B$	$\{A,B\}\ d_A = d_B$
LossySync	$A \dashrightarrow B$	$\{A\}$; $\{A,B\}\ d_A = d_B$
SyncDrain	A ⊳—⊲ B	$\{A,B\}$
AsyncDrain	A ⊳—‖—⊲ B	$\{B\}$; $\{A\}$
FIFO	A —▭→ B	$\{A\}\ d_A = 1$; $\{A\}\ d_A = 0$; $\{B\}\ d_B = 1$; $\{B\}\ d_B = 0$
Filter	A —∧∧→ B	$\{A\}\ \neg expr(d_A)$; $\{A,B\}\ expr(d_A) \wedge d_A = d_B$
Transform	A —▷→ B	$\{A,B\}\ d_B = f(d_A)$
Merger	A, B → C	$\{A,C\}\ d_A = d_C$; $\{B,C\}\ d_B = d_C$
Replicator	A → B, C	$\{A,B,C\}\ d_A = d_B = d_C$

Definition 3. *For two constraint automata $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \rightarrow_1, s_0^1)$ and $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \rightarrow_2, s_0^2)$ and the port synchronization function $\gamma : \mathcal{N} \rightarrow \mathcal{N}_1 \times \mathcal{N}_2$ with $\gamma_1 : \mathcal{N} \rightarrow \mathcal{N}_1$ and $\gamma_2 : \mathcal{N} \rightarrow \mathcal{N}_2$, the constraint automaton $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, called the γ-synchronous product of $\mathcal{A}_1$ and $\mathcal{A}_2$, is given by $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2 = (S_1 \times S_2, \mathcal{N}', \rightarrow, \langle s_0^1, s_0^2 \rangle)$ where $\mathcal{N}' = \mathcal{N}_1' \mid_\gamma \mathcal{N}_2'$ and the transition relation $\rightarrow$ is determined by the following rules:*

$$\frac{s_1 \xrightarrow{N_1, g_1}_1 t_1 \quad N_1 \subseteq \mathcal{N}_1'}{\langle s_1, s_2 \rangle \xrightarrow{N_1, g_1} \langle t_1, s_2 \rangle} \qquad \frac{s_2 \xrightarrow{N_2, g_2}_1 t_2 \quad N_2 \subseteq \mathcal{N}_2'}{\langle s_1, s_2 \rangle \xrightarrow{N_2, g_2} \langle s_1, t_2 \rangle}$$

and

$$\frac{s_1 \xrightarrow{N_1, g_1}_1 t_1 \quad s_2 \xrightarrow{N_2, g_2}_1 t_2 \quad \gamma_1^{-1}(N_1) = \gamma_2^{-1}(N_2)}{\langle s_1, s_2 \rangle \xrightarrow{N_1 \mid_\gamma N_2, \gamma(g_1 \wedge g_2)} \langle t_1, t_2 \rangle}.$$

In the above setting, for a port $n \in \mathcal{N}$, the idea is that the ports $x = \gamma_1(n) \in \mathcal{N}_1$ and $y = \gamma_2(n) \in \mathcal{N}_2$ synchronize. Thus, either x and y both have flow or x and y both have no flow, expressed as n having flow or no flow, respectively. The resulting automaton, the so-called synchronized product automaton $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, follows the flow of $\mathcal{A}_1$ and $\mathcal{A}_2$, based on the first two rules for the transition relation, but requires the flow on its ports in $\mathcal{N}$ to be agreed upon by both $\mathcal{A}_1$ and $\mathcal{A}_2$.

Constraint automata in their basic form are not expressive enough to capture all interesting behavior in Reo. In particular, they cannot express the behavior

of so-called context dependent channels. A basic example of such a channel is a LossySync channel that loses a data item only if the environment or subsequent channels are not ready to consume it. Numerous models have been proposed to overcome this and other problems. However, due to space limits we cannot provide their detailed discussion in this paper.

The problem of expressing the behavior of Reo circuits is orthogonal to the problem of estimating the end-to-end quality of the communication protocol that they implement. The existing semantic models, most notably, constraint automata, have been extended with the information to capture the QoS characteristics of the channels and their composition metrics [3,4]. However, these extensions assume that the QoS characteristics do not affect the behavior of a circuit and simply assign QoS labels to the transitions of the basic automata models. In the next section, we argue that data transfer delays are important for circuit behavior and accommodating them properly requires an appropriate formal model.

3 Motivation

As mentioned before, currently variants of constraint automata extended with labels representing QoS characteristics are used to give formal semantics to QoS-aware Reo. These automata are defined with the help of Q-algebra introduced initially by Chothia and Kleijn [6]. A Q-algebra is an algebraic structure $R = (C, \oplus, \otimes, ||, \mathbf{0}, \mathbf{1})$ where C is the domain of R and represents a set of QoS values. The operation $\oplus$ induces a partial order on the domain of R and is used to define a preferred value of a QoS dimension, $\otimes$ is an operator for the sequential channel composition, while $||$ is an operator for the parallel composition. For example, the Q-algebra corresponding to the circuit execution time is defined as follows: $(\mathbb{R}_{\geq} \cup \{\infty\}, min, +, max, \infty, 0)$. Taking into account this definition, a Quantitative CA (QCA) [3] is an extended CA $\mathcal{A} = (S, S_0, \mathcal{N}, E, R)$ where the transition relation E is a finite subset of $\cup_{N \in \mathcal{N}} S \times \{N\} \times DC(N) \times C \times S$ and $R = (C, \oplus, \otimes, ||, 0, 1)$ is a labeled Q-algebra with domain C.

QCA were introduced to enable the estimation of QoS of compound circuits given the QoS parameters of their constituent channels. However, as our example shown in Figure 1(a) illustrates, this model does not allow us to precisely compute the data transfer delays in synchronous regions. According to the definition of the QCA and Q-algebra for the execution time, the delay for the barrier synchronization connector equals $max(t_1, t_2, t_3, t_4, t_5)$ while the real data transfer time cannot be smaller than the sum of the delays of the two pairs of Sync channels composed sequentially. Assuming that the whole transaction does not finish before the SyncDrain channel destroys the data consumed through its source ports, we conclude that the delay equals $max(t_1 + max(t_2, t_3), t_4 + max(t_3, t_5))$.

As shown in [4], data transfer delays in Reo circuits can be computed given the information about its topology and the presumed dataflow semantics. However, currently there are no automata-based semantic models for Reo that supports such a computation in the compositional manner.

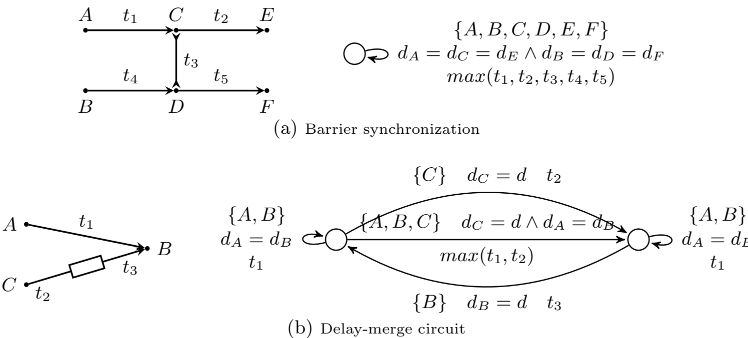


Fig. 1. Motivating examples

Another drawback of the constraint automata semantics for Reo is that it forces the synchronization of independent concurrent transactions with different durations. This problem arises from the fact that only one transition on constraint automaton can be enabled at the same time. Assuming that data transfer through a synchronous region of a circuit is not instantaneous as in the basic Reo model, dataflow on some other parts of the circuit can be initiated during this time. For example, the constraint automaton for the circuit shown in Figure 1(b) implies that while the FIFO channel accepts data through the port C, no other transition can be triggered. Imagine that the delay t_2 is much bigger than t_1. This means that the circuit will not transfer data through the channel $\mathsf{Sync}(A, B)$ until the port C finishes to accept data. However, the circuit should allow data transfer through the Sync channel at any time when the port B is not occupied. As this example illustrates, (Q)CA do not show all possible behaviors of Reo with data transfer delays.

4 Action Constraint Automata

In this section, we introduce a new model, called *action constraint automata*, that provides a valid semantic model for Reo coordination networks in the presence of time delays This model is essentially a labeled transition system (LTS) with data and synchronization constraints. However, in contrast to constraint automata in their classic form, we distinguish several kinds of actions which are triggered on channel ports to signal the state changes of the channel. Formally, an action constraint automaton is defined as follows:

Definition 4 (Action Constraint Automaton (ACA)). *An action constraint automaton $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ consists of a set of states S, a set of action names $\mathcal{N}$ derived from a set of port names $\mathcal{M}$ and a set of admissible action types $\mathcal{T}$, a transition relation $\rightarrow \subseteq S \times 2^{\mathcal{N}} \times DC \times S$, where DC is the set of data constraints over a finite data domain Data, and an initial state $s_0 \in S$.*

We introduce an injective function $act : \mathcal{M} \times \mathcal{T} \rightarrow \mathcal{N}$ to define action names for each pair of a port name and an action type observed on the port. For example, the function $act(m, \alpha) = \alpha \bullet m \,|\, m \in \mathcal{M}, \alpha \in \mathcal{T}$, where $\bullet$ is a standard lexical concatenation operator, can be used to obtain a set of unique action names given sets of distinctive Reo port names and types of observable actions.

Analogously to the constraint automata, we define the action synchronization function $\gamma : \mathcal{N} - > \mathcal{N}_1 x \mathcal{N}_2$ through a pair of injective functions $\gamma_1 : \mathcal{N} \rightarrow \mathcal{N}_1$, $\gamma_2 : \mathcal{N} \rightarrow \mathcal{N}_2$ from a new set of action names $\mathcal{N}$ into $\mathcal{N}_1$ and $\mathcal{N}_2$. Given such function, we can define the product operator for ACA.

Definition 5 (Product of Action Constraint Automata). *For two action constraint automata $\mathcal{A}_1 = (S_1, \mathcal{N}_1, \rightarrow_1, s_0^1)$ and $\mathcal{A}_2 = (S_2, \mathcal{N}_2, \rightarrow_2, s_0^2)$ and the action synchronization function $\gamma : \mathcal{N} \rightarrow \mathcal{N}_1 \times \mathcal{N}_2$ with $\gamma_1 : \mathcal{N} \rightarrow \mathcal{N}_1$ and $\gamma_2 : \mathcal{N} \rightarrow \mathcal{N}_2$, the action constraint automaton $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2$, called the γ-synchronization product of $\mathcal{A}_1$ and $\mathcal{A}_2$, is given by $\mathcal{A}_1 \bowtie_\gamma \mathcal{A}_2 = (S_1 \times S_2, \mathcal{N}_1' \,|_\gamma\, \mathcal{N}_2', \rightarrow, \langle s_0^1, s_0^2 \rangle)$ where the transition relation $\rightarrow$ is determined by the following rules:*

$$\frac{s_1 \xrightarrow{N_1, g_1}_1 t_1 \quad N_1 \subseteq \mathcal{N}_1'}{\langle s_1, s_2 \rangle \xrightarrow{N_1, g_1} \langle t_1, s_2 \rangle} \qquad \frac{s_2 \xrightarrow{N_2, g_2}_1 t_2 \quad N_2 \subseteq \mathcal{N}_2'}{\langle s_1, s_2 \rangle \xrightarrow{N_2, g_2} \langle s_1, t_2 \rangle}$$

and

$$\frac{s_1 \xrightarrow{N_1, g_1}_1 t_1 \quad s_2 \xrightarrow{N_2, g_2}_1 t_2 \quad \gamma_1^{-1}(N_1) = \gamma_2^{-1}(N_2)}{\langle s_1, s_2 \rangle \xrightarrow{N_1 |_\gamma N_2, \gamma(g_1 \wedge g_2)} \langle t_1, t_2 \rangle}.$$

Transitions where the set of actions N is non-empty are called *visible*, while transitions with the empty action-set are called *hidden*. In a hidden transition, none of the actions is visible and the data constraints appear as unknown from outside. We denote hidden transitions by the label τ. Such transitions can be witnessed only by the change of a state in an automaton. Taking this into account, the hiding operator on ACA is defined as follows:

Definition 6 (Action hiding). *The action hiding operator takes as input an ACA $\mathcal{A} = (S, \mathcal{N}, \rightarrow, s_0)$ and a non-empty set of actions $K \subseteq \mathcal{N}$. The result is an ACA $\mathtt{hide}(\mathcal{A}, K) = (S, \mathcal{N} \backslash K, \rightarrow, s_0)$ where*

- *$q \xrightarrow{N', g'}_K p$ iff there exists a transition $q \xrightarrow{N, g} p$ such that $N \backslash K \neq \emptyset$ and $g' = \bigvee_{\delta \in DA(K)} g[d_A / \delta.A | A \in K]$, where $g[d_A / \delta.A | A \in K]$ denotes the syntactic replacement of all occurrences of d_A in g for $A \in K$ with $\delta.A$.*
- *$q \xrightarrow{\tau}_K p$ iff there exists a transition $q \xrightarrow{N, g}_K p$ such that $N \backslash K = \emptyset$.*

A port hiding can be achieved by hiding of all actions observed on this port. In turn, a node hiding is the result of the hiding of all ports coincident on the node.

Note that constraint automata represent a subclass of action constraint automata with only one action observed on each port. This action represents the fact that the data flow through this port. The synchronization function used in the definition of constrain automata implies a renaming of joint channel/node ports while here it is used for renaming of actions that are observed simultaneously.

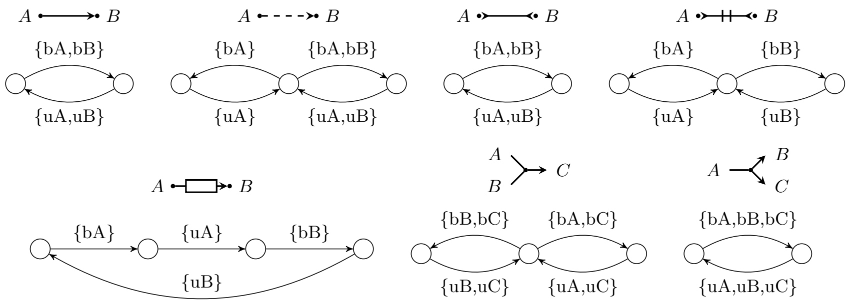


Fig. 2. Semantics of channels and nodes with port blocking

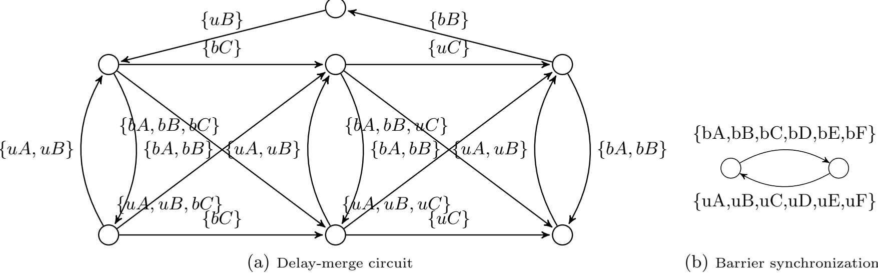
5 Dataflow Modeling

In this section, we introduce an ACA-based model for representing the semantics of Reo with data transfer delays.

Since some time is required by a channel for its internal coordination and to transfer data, it may happen that the channel is still busy while other requests arrive at the source ports of the circuit. There is no reason why the channels that are not busy at the moment should not process the arrived requests. However, as our motivating examples have shown, CA do not allow such behavior. To provide a more expressive model for Reo that fixes the aforementioned problem, we consider two actions, namely, a 'block' action and its dual an 'unblock' action which are used to establish port communication within a single transaction and release channel ports involved in such a transaction, respectively.

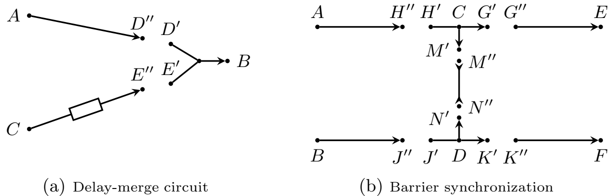
Table 5 shows the semantics of the basic Reo channels with presumable data transfer delays in terms of ACA for the set of action types $\mathcal{T}_1 = \{b, u\}$, where b stands for the 'block' and u stands for the 'unblock' actions. Since our focus is on synchronization constraints, we omit the data constraints in this figure to simplify the presentation. In their initial states, channels do not accept or dispense data. To show that the Sync channel with the source end A and the sink end B is ready to accept and dispense data, 'block' actions bA and bB occur simultaneously. After the data transfer is finished, the channel returns to its initial state when both ports are released and 'unblock' actions uA and uB are observed. For the LossySync channel the behavior is similar with the exception that the data can be lost after entering the channel. In this case, only the channel source port A is involved in the communication. For the SyncDrain channel we require that ports are blocked and unblocked simultaneously, while for the AsyncDrain only one of the source ends A or B can be involved in the communication at each particular moment in time. Finally, for the empty FIFO channel, first the data is stored in the buffer through the source port A, then the buffer is emptied through the sink port B.

Figure 3 shows the semantics of the delay-merge circuit and the barrier synchronization of Figure 1 obtained using the product and hiding operators on the ACA with the set of action types $\mathcal{T}_1$. Since in our definitions of CA and ACA we



(a) Delay-merge circuit (b) Barrier synchronization

Fig. 3. Semantics of the delay-merge circuit with port blocking



(a) Delay-merge circuit (b) Barrier synchronization

Fig. 4. Motivating examples: decomposed circuits

require all port names to be different, we apply these operators to the channels and nodes in the decomposed versions of these circuits shown in Figure 4. Observe that after blocking the port C in the delay-merge circuit, the system can trigger transitions defined by the actions $\{bA, bB\}$ and $\{uA, uB\}$. This means that the Sync channel can transfer data while the FIFO channel is writing data into its buffer. Thus, our new ACA semantic model resolves the problem of synchronization of independent concurrent transactions with different durations manifested by the CA model.

The automaton for the barrier synchronization consists of two states that represent the situations where all channels in the synchronous region are free and where all channels are transferring data. The circuit can stay in the second state for the duration of time needed to finish the data transfer through all five synchronous channels. However, the model does not show the actual flow of data within the circuit. Thus, we cannot use this model for computing time delays in synchronous circuits given delays for its individual channels.

To solve this problem, in addition to the 'block' and 'unblock' actions, we introduce 'start' and 'finish' actions which are used to represent the start and the end of dataflow through a blocked channel port. Thus, we use the set of action types $\mathcal{T}_2 = \{b, s, f, u\}$, where b stands for the 'block', s for the 'start', f for the 'finish' and u for the 'unblock' action types. The sequence of the aforementioned four actions is observed on each Reo port. Before the start of each transition, ports participating in this transition must be blocked. Then, the data transfer

starts. After some time t, which represents the delay in the channel, the 'finish' action occurs to signal that the data transfer is over. Finally, the 'unblock' action releases the channel port, subsequent to which it can be coopted to perform another communication. The time between a 'block' action and a subsequent 'start' action on the same port represents the overhead necessary for the set-up of the internal coordination before the data transfer can happen. Analogously, the time between a 'finish' and an 'unblock' represents the overhead of dismantling the data transfer set-up. Table 2 shows the semantics of the basic Reo channels with explicit modeling of internal coordination and dataflow within each channel. After blocking actions have occurred in the Sync channel, both its ports start to accept data. This is represented by the simultaneous occurrence of the actions sA and sB. Similarly, after the data transfer is finished, actions fA and fB are observed. For the SyncDrain channel, as usual, we require that its ports are blocked and unblocked simultaneously, while the actual data transfer through the two ports start and end independently, i.e., all interleavings of action pairs (sA, fA) and (sB, fB) are allowed. In principle, it is also possible to consider more restricting versions of the SyncDrain channel where both source ports must synchronize on starting and/or finishing of their data transfer.

The semantics of the Merger and the Replicator nodes is defined in a similar way. We assume that, in contrast to channels, the data transfer through a node is instantaneous, i.e., dataflow starts and finishes at the same time. For the scenarios where the time for data replication is significant and cannot be neglected, automata with two different transitions to signal the start and the end of dataflow should be used.

By synchronizing 'finish' actions observed on sink ends with 'start' actions observed on the source ends, we can model sequential flow of data in the synchronous regions. Given two action constraint automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for each pair $X \in \mathcal{M}_1, Y \in \mathcal{M}_2$ of joint ports, where X is a sink port, and Y is a source port, the following pairs of actions happen synchronously:

$$\{(act(X, b), act(Y, b)),\ (act(X, u), act(Y, u)),\ (act(X, f), act(Y, s))\}.$$

This approach is compliant with the two principles introduced in [4], namely, that (i) a data-flow in a channel takes place from its input port to its output port, and (ii) mixed nodes receive and send data instantaneously.

Figure 5 shows the ACA with the set of action types $\mathcal{T}_2$ for the delay-merge circuit obtained as a product of ACA for two channels and the merge node with an action synchronization function defined by the following set of mappings:

$$\{(bD'', bD') \to bD,\ (uD'', uD') \to uD,\ (fD'', sD') \to fD,$$
$$(bE'', bE') \to bE,\ \ (uE'', uE') \to uE,\ (fE'', sE') \to fE\}.$$

Observe that, similarly to the previous example, in any state where port C is occupied (blocked, started to or finished with the transfer of data, but not yet unblocked), the Sync channel can be involved in an independent communication.

Table 2. Semantics of channels and nodes with explicit dataflow

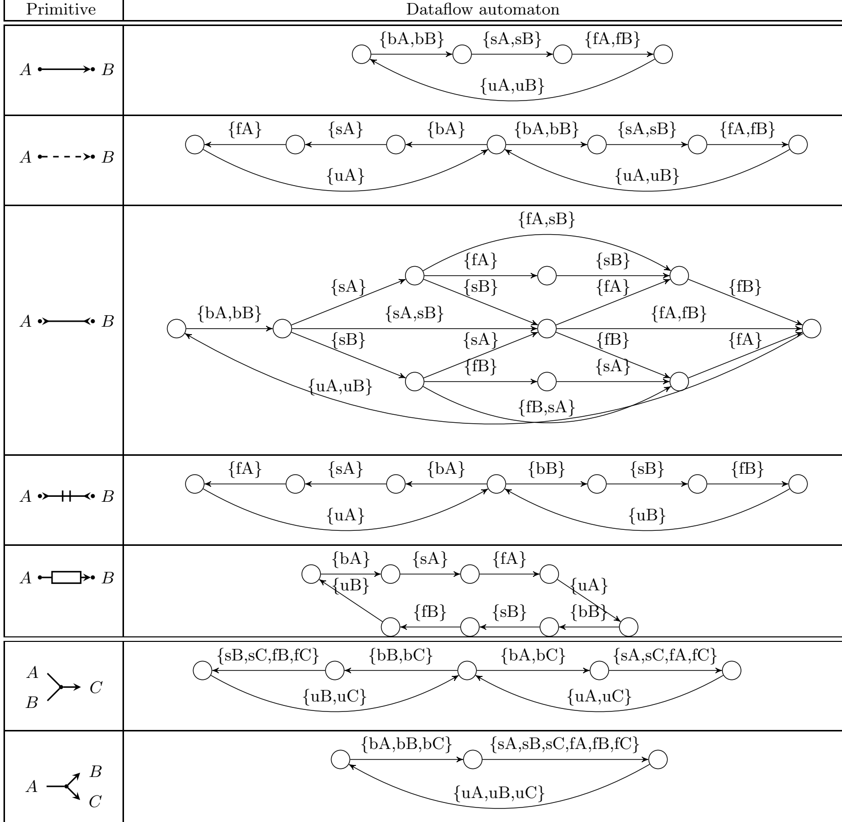


Figure 6 shows the ACA for the barrier synchronization circuit obtained using the action synchronization function defined by the following mappings:

$$\begin{aligned}
\{(bH'', bH') \to bH,\ (uH'', uH') \to uH,\ (fH'', sH') \to fH,\\
(bG', bG'') \to bG,\ (uG', uG'') \to uG,\ (fG', sG'') \to sG,\\
(bM', bM'') \to bM,\ (uM', uM'') \to uM,\ (fM', sM'') \to sM,\\
(bN', bN'') \to bN,\ (uN', uN'') \to uN,\ (fN', sN'') \to sN,\\
(bJ'', bJ') \to bJ,\ (uJ'', uJ') \to uJ,\ (fJ'', sJ') \to fJ,\\
(bK', bK'') \to bK,\ (uK', uL'') \to uK,\ (fK', sK'') \to sK\}
\end{aligned}$$

and the set of hidden actions

$$\{fH', sH'', fG'', sG', sM', fM'', sN', fN'', fJ', sJ'', sK', fK''\}.$$

In this model, after blocking all ports, the source ports A and B start to accept data (either separately or simultaneously). Similarly, labels of further transitions show on which ports the dataflow starts and finishes. Observe that the end

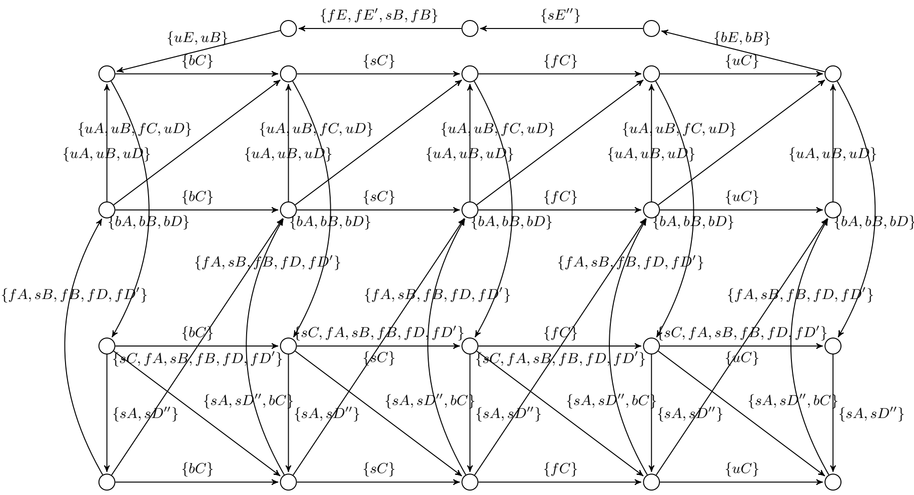


Fig. 5. Semantics of the delay-merge circuit with explicit dataflow

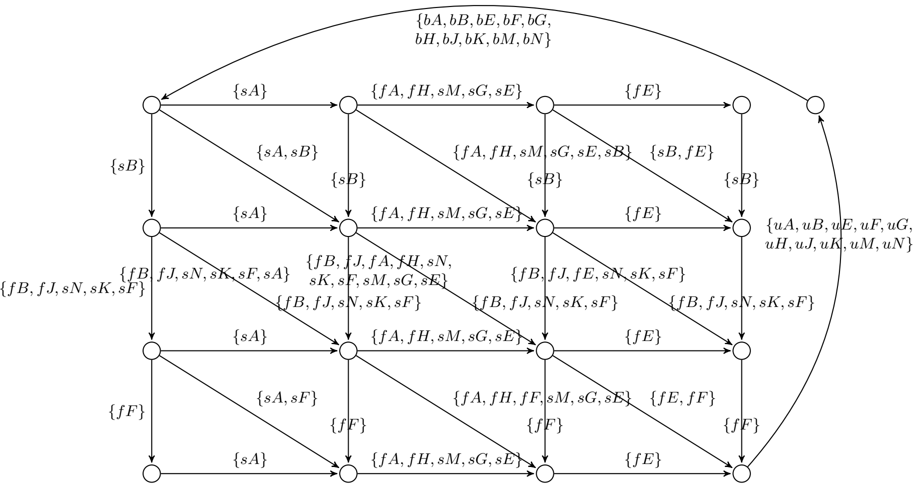


Fig. 6. Semantics of the barrier synchronization circuit with explicit dataflow

of dataflow on ports preceding the SyncDrain (external ports A and B, and internal ports H and J) coincides with the start of the flow on ports following the SyncDrain (external ports E and F, and internal ports M, N, G and K). Thus, this model is capable of capturing the stepwise dataflow progress through synchronous regions.

Among all the states of these automata we may be interested to locate states in which all channels are idle and free to communicate. Formally, such states are

characterized by the condition $\forall A \in \mathcal{M}, act(A, b) \in N[r] \Rightarrow act(A, u) \in N[r]$, where $N[r] = \bigcup N_i \mid s_i \xrightarrow{N_i, d_i} s_{i+1}$ is a set of actions of some automaton run $r = s \xrightarrow{N_0, d_0} s_1 \xrightarrow{N_1, d_1} s_2 \xrightarrow{N_2, d_2} s_3 ....$ Such states correspond to network configurations defined by the basic CA.

6 Model Analysis and Tool Support

The goal of the introduced semantic model for Reo is to provide a sound mathematical basis for the implementation of analysis tools. The set of potentially useful tools includes but is not limited to converters that generate the automata-based models given graphical Reo circuits, model checking tools able to verify the validity of system properties expressed in some kind of formal logic, simulation engines that allow us to validate and evaluate the performance of a model, and model-based code and test generation tools. The development of such tools from scratch is far from trivial and very time consuming. An alternative approach is to convert our model to a format acceptable by existing analysis tools. To enable model checking of Reo, we generally rely on the `mCRL2` framework.

`mCRL2` is a specification language based on the process algebra ACP. The basic notion in `mCRL2` is the action. Actions represent atomic events and can be parameterized with data. Actions in `mCRL2` can be synchronized using the synchronization operator $|$. Synchronized actions are called multiactions. Processes are defined by process expressions, which are compositions of actions and multiactions using a number of operators. The basic operators include (i) *deadlock* or *inaction* δ, (ii) *alternative composition* $p + q$, (iii) *sequential composition* $p \cdot q$, (iv) *conditional* operator or *if-then-else* construct $c \rightarrow p \diamond q$ where c is a boolean expression, (v) *summation* $\Sigma_{d:D}\, p$ used to quantify over a data domain D, (vi) *at* operator $a@t$ indicating that multiaction a happens at time t, (vii) *parallel composition* $p \parallel q$ yielding interleavings of the actions in p and q, (viii) *encapsulation* $\partial_H(p)$, where H is a set of action names that are not allowed to occur, (ix) *renaming* operator $\rho_R(p)$, where R is a set of renamings of the form $a \rightarrow b$ and (x) *communication* operator $\Gamma_C(p)$, where C is a set of communications of the form $a_0|...|a_n \mapsto c$, which means that every group of actions $a_0|...|a_n$ within a multiaction is replaced by c. Moreover, the `mCRL2` language provides a number of built-in datatypes (e.g., boolean, natural, integer) with predefined standard arithmetic operations and a datatype definition mechanism to declare custom types (called also sorts).

The `mCRL2` toolset includes a tool for converting `mCRL2` code into a linear process specification (LPS), which is a compact symbolic representation of LTS to speed up subsequent manipulations, a tool for generating explicit LTS from LPS, tools for optimizing and visualizing LTS, and many other useful facilities. For model checking, system properties are specified as formulae in a variant of the modal μ-calculus extended with regular expressions, data and time. In combination with an LPS such a formula is transformed into a parameterized boolean equation system and can be solved with the appropriate tools from the toolset. Analysis at the level of LTS, in particular, deadlock detection or checking

Table 3. mCRL2 encoding for channels and nodes

$$\begin{aligned}
\mathsf{Sync} &= bA|bB \cdot sA|sB \cdot fA|fB \cdot uA|uB \cdot \mathsf{Sync}\\
\mathsf{LossySync} &= (bA|bB \cdot sA|sB \cdot fA|fB \cdot uA|uB + bA \cdot sA \cdot fA \cdot uA) \cdot \mathsf{LossySync}\\
\mathsf{SyncDrain} &= bA|bB \cdot (\\
&\quad sA \cdot (sB \cdot (fA \cdot fB + fB \cdot fA + fA|fB) + fA \cdot sB \cdot fB + sB|fA \cdot fB)+\\
&\quad sB \cdot (sA \cdot (fA \cdot fB + fB \cdot fA + fA|fB) + fB \cdot sA \cdot fA + sA|fB \cdot fA)+\\
&\quad sA|sB \cdot (fA \cdot fB + fB \cdot fA + fA|fB)) \cdot uA|uB \cdot \mathsf{SyncDrain}\\
\mathsf{AsyncDrain} &= (bA \cdot sA \cdot fA \cdot uA + bB \cdot sB \cdot fB \cdot uB) \cdot \mathsf{AsyncDrain}\\
\mathsf{FIFO} &= \mathit{isEmpty}(f) \rightarrow bA \cdot sA \cdot fA \cdot uA \cdot \mathsf{FIFO}(\mathit{full})\\
&\qquad \diamond\ bB \cdot bB \cdot sB \cdot fB \cdot uB \cdot \mathsf{FIFO}(\mathit{empty})
\end{aligned}$$

$$\begin{aligned}
\mathsf{Merger} &= (bA|bC \cdot sA|sC|fA|fC.uA|uC + bB|bC \cdot sB|sC|fB|fC \cdot uB|uC) \cdot \mathsf{Merger}\\
\mathsf{Replicator} &= bA|bB|bC \cdot sA|sB|sC \cdot fA|fC \cdot uA|uC \cdot \mathsf{Replicator}
\end{aligned}$$

of the presence or absence of certain actions, is also possible. A detailed overview can be found at the mCRL2 web site[2].

We employed the mCRL2 toolset to generate state spaces for graphical Reo circuits and further model check them. mCRL2 models for Reo circuits are generated in the following way [7]: observable actions (i.e., dataflow on the channel ends in the basic CA model) are represented as atomic actions, while data items observed at these ports are modeled as parameters of these actions. Analogously, we introduce a process for every node and actions for all channel ends meeting at the node. A global custom sort *Data* and the mCRL2 summation operator are used to model the input data domain and iterate over it while specifying data constraints imposed by channels.

The availability of the synchronization operator and multications in mCRL2 makes the translation of CA and ACA to the process algebra mCRL2 straightforward: we simply synchronize the joint ports in CA and the simultaneously observed actions in ACA. Table 3 shows the mCRL2 encodings for the basic Reo channels and nodes according to the semantic model introduced in this paper. Since data support in the new translation is analogous to the case of the CA-based translation [7], we omit its discussion here and for simplicity show only the data-agnostic mapping. Note that the expression for the SyncDrain channel in the table is equivalent to

$$\mathsf{SyncDrain} = bA|bB \cdot ((sA \cdot fA)||(sB \cdot fB)) \cdot uA|uB \cdot \mathsf{SyncDrain};$$

However, the use of the parallel operator in mCRL2 is restricted because of the difficulties to linearize processes where such an operator occurs in the scope of the sequential, alternative, summation or synchronization operators.

As in the CA approach, we construct nodes compositionally out of the Merger and the Replicator primitives. Given process definitions for all channels and nodes, a joint process that models the complete Reo connector is built by forming a parallel composition of these processes and synchronizing the actions for the coinciding channel/node ends. Optionally, the mCRL2 hiding operator can

[2] www.mcrl2.org/

be employed for abstracting the flow in internal nodes. Channel/node end synchronization is performed using two of the mCRL2 operators: communication and encapsulation. For minimizing intermediate state spaces while generating the mCRL2 specification, we exploit the structure of the circuit and build the process for the whole Reo connector in a stepwise fashion. In [5], we show that the operational semantics of the mCRL2 specification obtained in this way is equivalent to the CA semantics of the Reo connector. This result applies to ACA as well.

7 Conclusions

In this paper, we discussed the formal semantic models for the channel-based coordination language Reo in the presence of coordination and data transfer delays. We argued that the existing semantic models do not reflect all possible behaviors in such circuits and are not suitable for the computation of end-to-end time delays in Reo circuits. To fix these problems, we proposed a more expressive model, *action constraint automata*, which represent the behavior of a circuit in terms of actions observed on its ports. The new model distinguishes transactional aspects of Reo from dataflow modeling, which is useful for the implementation of animation and simulation tools for Reo as well as the implementation of Reo-based service interaction protocols.

The presented work is a first step toward enabling performance analysis for service compositions and process models specified in Reo. We are going to define the quantitative version of the ACA and develop algorithms for computing time delays in the circuits, which are rather straightforward, but are not discussed here due to space limitation. We also plan to consider circuits with stochastic delays and develop a theory of quality preserving substitutability of channel-based connectors.

References

1. Arbab, F.: Reo: A channel-based coordination model for component composition. Mathematical Structures in Computer Science 14, 329–366 (2004)
2. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling Component Connectors in Reo by Constraint Automata. Science of Computer Programming 61, 75–113 (2006)
3. Arbab, F., Chothia, T., Sun, M., Moon, Y.J.: Component connectors with QoS guarantees. In: Murphy, A.L., Vitek, J. (eds.) COORDINATION 2007. LNCS, vol. 4467, pp. 286–304. Springer, Heidelberg (2007)
4. Arbab, F., Chothia, T., van der Mei, R., Sun, M., Moon, Y., Verhoef, C.: From coordination to stochastic models of QoS. In: Field, J., Vasconcelos, V.T. (eds.) COORDINATION 2009. LNCS, vol. 5521, pp. 268–287. Springer, Heidelberg (2009)
5. Kokash, N., Krause, C., de Vink, E.: Verification of context-dependent channel-based service models. In: de Boer, F.S. (ed.) FMCO 2009. LNCS, vol. 6286, pp. 21–40. Springer, Heidelberg (2010)
6. Chothia, T., Kleijn, J.: Q-automata: Modelling the resource usage of concurrent components. In: Proc. FOCLASA 2006, pp. 79–94 (2007)
7. Kokash, N., Krause, C., de Vink, E.: Data-aware design and verification of service composition with Reo and mCRL2. In: Proc. of SAC 2010, pp. 2406–2413. ACM Press, New York (2010)

Compensable WorkFlow Nets

Fazle Rabbi, Hao Wang, and Wendy MacCaull*

Centre for Logic and Information
St. Francis Xavier University
{x2010mcf,hwang,wmaccaul}@stfx.ca

Abstract. In recent years, Workflow Management Systems (WfMSs) have been studied and developed to provide automated support for defining and controlling various activities associated with business processes. The automated support reduces costs and overall execution time for business processes, by improving the robustness of the process and increasing productivity and quality of service. As business organizations continue to become more dependant on computarized systems, the demand for reliability has increased. The language *t-calculus* [8] was developed to aid in the creation and verification of compensable systems. Motivated by this we define *Compensable WorkFlow nets* (CWF-nets) and introduce a graphical modeling language *Compensable Workflow Modeling Language* (CWML). We present a case study, using CWML to model a real world scenario, translate the resulting CWF-net into DVE (the input language of the DiVinE model checker) and verify properties of interest.

1 Introduction

A traditional system which consists of ACID (Atomicity, Consistency, Isolation, Durability) transactions cannot handle *long lived transaction* as it has only a flow in one direction. A long lived transaction system is composed of sub-transactions and therefore has a greater chance of partial effects remaining in the system in the presence of some failure. These partial effects make traditional rollback operations infeasible or undesirable. A *compensable transaction* is a type of transaction whose effect can be semantically undone even after it has committed [1]. A compensable transaction has two flows: a forward flow and a compensation flow. The forward flow executes the normal business logic according to system requirements, while the compensation flow removes all partial effects by acting as a backward recovery mechanism in the presence of some failure.

The concept of a compensable transaction was first proposed by Garcia-Molina and Salem [2], who called this type of long-lived transaction, a *saga*. A saga can be broken into a collection of sub-transactions that can be interleaved in any way with other sub-transactions. This allows sub-transactions to commit prior to the completion of the whole saga. As a result, resources can be released earlier and the possibility of deadlock is reduced. If the system needs to rollback in case of some failure, each sub-transaction executes an associated compensation

* Three authors contributed equally to this paper.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 122–137, 2010.
© Springer-Verlag Berlin Heidelberg 2010

to semantically undo the committed effects of its own committed transaction. Bruni et al. worked on long running transactions and compared Sagas with CSP [3] in the modelling of compensable flow composition [4].

In recent years, He et al. [5-8] have developed a specification for compensable transactions in order to provide increased system reliability. Ideally, such compensable transactions would be used to model a larger computer system, by composing several compensable sub-transactions to provide more complex functionality. These transactions would provide backward recovery if an intermediate error were to occur in the larger system. The result of this research was the creation of a transactional composition language, the t-calculus [8].

Workflow management systems (WfMS) provide an important technology for the design of computer systems which can improve information system development in dynamic and distributed organizations. Motivated by the petri net [9] and t-calculus [8] formalisms and the graphical representations underlying the YAWL [11] and ADEPT2 [12] modeling languages, we define *Compensable WorkFlow nets* (CWF-nets) and develop a new workflow modeling language, the *Compensable Workflow Modeling Language* (CWML). In addition, we present a verification method for CWF-nets using model checkers. Using this approach, both requirements and behavioural dependencies of transactions (specified in a temporal logic) can be verified efficiently. DiVinE [14], a well known distributed model checker, is used in the case study.

Section 2 provides some background information. The CWF-nets and the graphical modeling language, CWML, are defined in section 3. Section 4 presents the verification method. A case study is provided in section 5, and section 6 concludes the paper.

2 Background

Petri nets [9], developed by Petri in the early 1960s, is a powerful formalism for modeling and analyzing process.

Definition 1. *A petri net is a 5-tuple,* $PN = (P, T, F, W, M_0)$ *where:*

- $P = \{p_1, p_2, ..., p_m\}$ *is a finite set of places,*
- $T = \{t_1, t_2, ..., t_n\}$ *is a finite set of transitions,*
- $F \subseteq (P \times T) \cup (T \times P)$ *is a set of arcs (flow relation),*
- W*:* $F \rightarrow \{1,2,3,...\}$ *is a weight function,*
- M_0*:* $P \rightarrow \{0,1,2,3,...\}$ *is the initial marking,*
- $P \cap T = \phi$ *and* $P \cup T \neq \phi$*.*

A petri net structure $N = (P, T, F, W)$ *without any specific initial marking is denoted by N.*

Places may contain tokens and the distribution of tokens among the places of a petri net determine its *state* (or *marking*).

The coloured petri Net [10] is an extension to the petri net, where the tokens are valued and typed so that they can be distinguished.

Definition 2. *A coloured petri net is a 5-tuple, $CPN = (P, T, C, IN, OUT)$ where:*

- *P is a finite set of places,*
- *T is a finite set of transitions,*
- *C is a colour function such that $C : P \cup T \rightarrow$ Non-empty sets of colours,*
- *IN and OUT are functions with domain $(P \times T)$ such that for all $(p,t) \in P \times T$, $IN(p,t)$, $OUT(p,t)$:$C(p) \rightarrow [C(t) \rightarrow N]_f$, where $[C(t) \rightarrow N]_f$ denotes the set of all total functions g from C(t) to N with the support $a \in C(t)$:$g(a) \neq 0$ finite.*

A compensable transaction refers to a transaction with the capability to withdraw its result after its commitment, if an error occurs. A compensable transaction is described by its external state. There is a finite set of eight independent states, called *transactional states*, which can be used to describe the external state of a transaction at any time. These transactional states include `idle` (`idl`), `active` (`act`), `aborted` (`abt`), `failed` (`fal`), `successful` (`suc`), `undoing` (`und`), `compensated` (`cmp`), and `half-compensated` (`hap`), where `idl`, `act`, etc are the abbreviated forms. Among the eight states, `suc`, `abt`, `fal`, `cmp`, `hap` are the terminal states. The transition relations of the states are illustrated in Fig. 1.

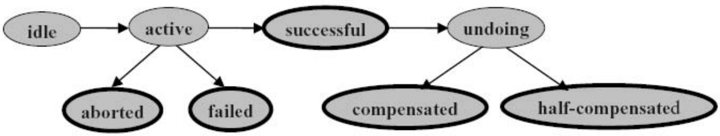


Fig. 1. State transition diagram of compensable transaction [1]

Before activation, a compensable transaction is in the `idle` state. Once activated, the transaction eventually moves to one of five terminal states. A `successful` transaction has the option of moving into the `undoing` state. If the transaction can successfully undo all its partial effects it goes into the `compensated` state, otherwise it goes into the `half-compensated` state. An ordered pair consisting of a compensable transaction and its state is called an *action*. Actions are the key to describing the behavioural dependencies of compensable transactions. In [1], five binary relations were proposed to define the constraints applied to actions on compensable transactions. Informally the relations are described as follows, where both a and b are actions:

1. $a < b$: only a can fire b.
2. $a \prec b$: b can be fired by a.
3. $a \ll b$: a is the precondition of b.
4. $a \leftrightarrow b$: a and b both occur or both not.
5. $a \nleftrightarrow b$: the occurance of one action inhibits the other.

The transactional composition language, t-calculus [8], was proposed to create reliable systems composed of compensable transactions. In addition, it provides flexibility and specialization, commonly required by business process management systems, with several alternative flows to handle the exceptional cases.

The syntax of t-calculus is made up of several operators which perform compositions of compensable transactions. Table 1 shows eight binary operators, where S and T represent arbitrary compensable transactions. These operators specify how compensable transactions are coupled and how the behaviour of a certain compensable transaction influences that of the other. The operators are discussed in detail in [1,5,6] and described in section 3.

Table 1. t-calculus syntax

Sequential Composition	$S\ ;\ T$	Parallel Composition	$S \mid\mid T$
Internal Choice	$S \sqcap T$	Speculative Choice	$S \otimes T$
Alternative Forwarding	$S \rightsquigarrow T$	Backward Handling	$S \trianglerighteq T$
Forward Handling	$S \rhd T$	Programmable Composition	$S \divideontimes T$

3 Workflow with Compensable Transactions

A workflow consists of steps or tasks that represent a work process. We extend the task element with the concept of compensable transaction, and accordingly adapt t-calculus for the composition of a compensable workflow. In this section, first, we present the formal syntax and semantics of Compensable Workflow nets; second, we present the graphical representation of the language, and finally, we give formal analysis of the language.

3.1 Compensable Workflow Nets

We build a compensable workflow using a group of tasks connected with operators; a task handles a unit of work and operators indicate the nature of the flow (forward and/or backward) relations between tasks. In particular, we define a colored petri net part for each task, therefore a CWF-net corresponds to a complete colored petri net. We use the colored petri net because in reality, a workflow handles several job instances (with different progress) at the same time so that the token of each color can be used to represent a specific job instance.

An atomic task [13] is an indivisible unit of work. Atomic tasks can be either compensable or uncompensable. We follow the general convention [13] of assuming that if activated, an atomic uncompensable task always succeeds.

Definition 3. *An atomic uncompensable task t is a tuple (s, P_t) such that:*

- *P_t is a petri net, as shown in Fig. 2;*
- *s is a set of unit states {idle, active, successful}; the unit state idle indicates that there is no token in P_t; the unit states active and successful indicate that there is a token in the place p_act and p_succ respectively;*

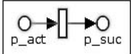


Fig. 2. Petri net representation of an uncompensable task

Remark that the unit states of a task are different from the state (marking) of a petri net. In addition, the unit state of a task is token-specific, i.e., a task is in a unit state for token(s) of a specific color and it may be in a different state for token(s) of another color.

Definition 4. *An atomic compensable task* t_c *is a tuple* (s_c, P_{t_c}) *such that:*

- P_{t_c} *is a petri net; as shown in Fig. 3;*
- *s is a set of unit states {idle, active, successful, undoing, aborted, failed}; the unit state idle indicates that there is no token in* P_{t_c}*; the other unit states indicate that there is a token in the relevant place;*

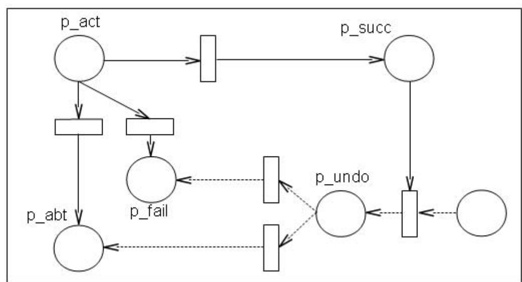


Fig. 3. Petri net representation of a compensable task

The compensation flow is denoted by the dotted arcs. Note that an atomic compensable task does not have the *compensated* and *half-compensated* states. We drop these states as their semantics overlap with the aborted and failed states. The *compensated* state means successful compensation, and corresponds to *aborted* after *successful*; the *half-compensated* state means error during compensation, resulting in data inconsistencies (i.e., remained partial effects); it corresponds to *failed* after *successful*.

The task t_c transits to the unit state of *active* after getting a token in `p_act`. The token can move to either `p_succ`, `p_abt` or `p_fail` representing unit states *successful*, *aborted* or *failed* respectively. The unit state *aborted* indicates an error occurred performing the task and the effects can be successfully removed. The backward (compensation) flow is started from this point. On the other hand, the unit state *failed* indicates that the error cannot be removed successfully and the partial effect will remain in the system unless there is an exception handler. Note that t_c can transit to the unit states *aborted* or *failed* either before or after the unit state *successful*.

A compensable task can be composed with other compensable tasks using t-calculus operators.

Definition 5. *A compensable task (ϕ_c) is recursively defined by the following well-formed formula: (Adapted from [1])*

$$\phi_c = t_c \mid (\phi_c \odot \phi_c)$$

where t_c is an atomic compensable task, and $\odot \in \{;, ||, \sqcap, \otimes, \rightsquigarrow, \trianglerighteq, \triangleright, \circledast\}$ is a t-calculus operator defined as follows:

- $\phi_{c_1}\ ;\ \phi_{c_2}$*: ϕ_{c_2} will be activated after the successful completion of ϕ_{c_1},*
- $\phi_{c_1} \mid\mid \phi_{c_2}$*: ϕ_{c_1} and ϕ_{c_2} will be executed in parallel. If either of them (ϕ_{c_1} or ϕ_{c_2}) is aborted, the other one will also be aborted,*
- $\phi_{c_1} \sqcap \phi_{c_2}$*: either ϕ_{c_1} or ϕ_{c_2} will be activated depending on some internal choice,*
- $\phi_{c_1} \otimes \phi_{c_2}$*: ϕ_{c_1} and ϕ_{c_2} will be executed in parallel. The first task that reaches the goal will be accepted and the other one will be aborted,*
- $\phi_{c_1} \rightsquigarrow \phi_{c_2}$*: ϕ_{c_1} will be activated first to acieve the goal, if ϕ_{c_1} is aborted, ϕ_{c_2} will be executed to achieve the goal,*
- $\phi_{c_1} \trianglerighteq \phi_{c_2}$*: if ϕ_{c_1} fails during execution, ϕ_{c_2} will be activated to remove the partial effects remaining in the system. ϕ_{c_2} terminates the flow after successfully removing the partial effects,*
- $\phi_{c_1} \triangleright \phi_{c_2}$*: if ϕ_{c_1} fails, ϕ_{c_2} will be activated to remove the partial effects. ϕ_{c_2} resumes the forward flow to achieve the goal,*
- $\phi_{c_1} \circledast \phi_{c_2}$*: if ϕ_{c_1} needs to undo its effect, the compensation flow will be redirected to ϕ_{c_2} to remove the effects.*

Any task can be composed with uncompensable and/or compensable tasks to create a new task.

Definition 6. *A task (ϕ) is recursively defined by the following well-formed formula:*

$$\phi = t \mid (\phi_c) \mid\mid (\phi \ominus \phi)$$

where t is an atomic task, ϕ_c is a compensable task, and $\ominus \in \{\wedge, \vee, \times, \bullet\}$ is a control flow operator defined as follows:

- $\phi_1 \wedge \phi_2$*: ϕ_1 and ϕ_2 will be executed in parallel,*
- $\phi_1 \vee \phi_2$*: ϕ_1 or ϕ_2 or both will be executed in parallel,*
- $\phi_1 \times \phi_2$*: exclusively one of the task (either ϕ_1 or ϕ_2) will be executed,*
- $\phi_1 \bullet \phi_2$*: ϕ_1 will be executed first then ϕ_2 will be executed.*

A subformula of a well-formed formulae is also called a *subtask*. We remark that if T_1 and T_2 are compensable tasks, then $T_1;T_2$ denotes another compensable task while $T_1 \bullet T_2$ denotes a task consisting of two distinct compensable subtasks. Any task which is built up using any of the operators $\{\wedge, \vee, \times, \bullet\}$ is deemed as uncompensable.

In order for the underlying petri net to be complete, we add a pair of split and join routing tasks for operators including $\wedge$, $\vee$, $\times$, $||$, $\sqcap$, $\otimes$, and $\rightsquigarrow$ and we give

their graphical representation in section 3.2. Each of these routing tasks has a corresponding petri net representation, e.g., for the speculative choice operator $\phi_{c_1} \otimes \phi_{c_2}$, the split routing task will direct the forward flow to ϕ_{c_1} and ϕ_{c_2}; the task that performs its operation first will be accepted and the other one will be aborted.

Now we can present the formal definition of Compensable WorkFlow nets (CWF-nets):

Definition 7. *A Compensable Workflow net (CWF-net) C_N is a tuple (*i*, *o*, T, T_c, F) such that:*

- *$\boldsymbol{i}$ is the input condition,*
- *$\boldsymbol{o}$ is the output condition,*
- *T is a set of tasks,*
- *$T_c \subseteq T$ is a set of compensable tasks, and $T \backslash T_c$ is a set of uncompensable tasks,*
- *$F \subseteq (\{\boldsymbol{i}\} \times T) \cup (T \times T) \cup (T \times \{\boldsymbol{o}\})$ is the flow relation (for the net),*
- *the first atomic compensable task of a compensable task is called the initial subtask; the backward flow from the initial subtask is directed to the output condition,*
- *every node in the graph is on a directed path from $\boldsymbol{i}$ to $\boldsymbol{o}$.*

If a compensable task aborts, the system starts to compensate. After the full compensation, the backward flow reaches the initial subtask of the compensable task and the flow terminates, as the backward flow of an initial task of compensable tasks is connected with the output condition.

The reader must distinguish between the flow relation (F) of the net, as above and the internal flows of the atomic (uncompensable and compensable) tasks.

A CWF-net such that $T_c = T$ is called a *true Compensable workflow net* (CWF_t-net).

3.2 Graphical Representation of CWF-Nets

We first present graphical representation of t-calculus operators, then present the contruction principles for modeling a compensable workflow. Fig. 4 gives graphical represetation of tasks. Some of the operators are described in this section.

Sequential Composition. Two compensable tasks ϕ_{c_1} and ϕ_{c_2} can be composed with sequential composition as shown in Fig. 4, which represents the formula ϕ_{c_1} ; ϕ_{c_2}. Task ϕ_{c_2} will be activated only when task ϕ_{c_1} finishes its operations successfully. For the compensation flow, when ϕ_{c_2} is aborted, ϕ_{c_1} will be activated for compensation, i.e., to remove its partial effects.

Recall that in CWF-nets, we drop the *compensated* and *half-compensated* states because their semantics overlap with the *aborted* and *failed* states; therefore, we do not consider the two states in the behavioural dependencies. The following two basic dependencies [1] describe behaviour of sequential composition: **i)** $(\phi_{c_1}, suc) < (\phi_{c_2}, act)$; **ii)** $(\phi_{c_2}, abt) \prec (\phi_{c_1}, und)$;.

Theorem 1. *The above dependencies hold in the petri net representation (as in Fig. 3) of sequential composition.*

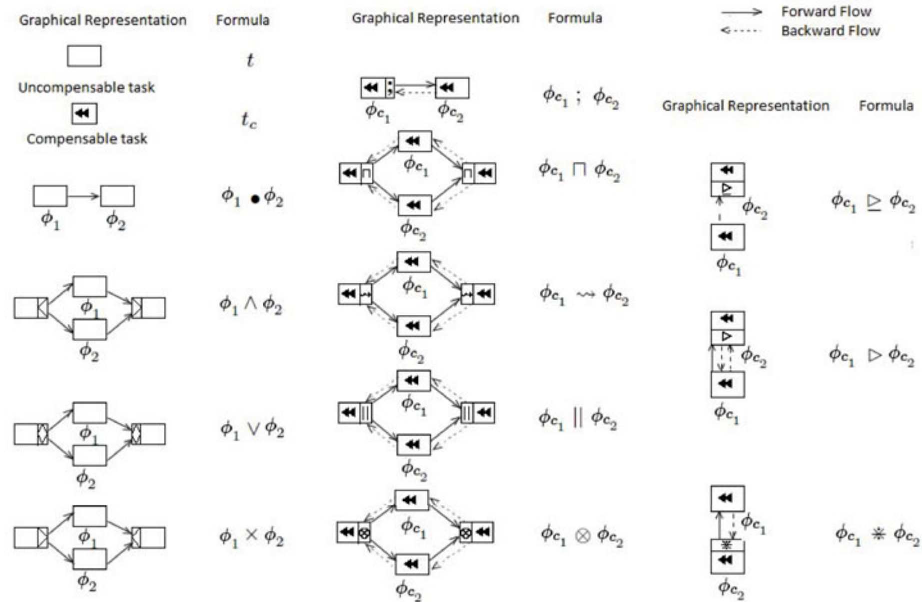


Fig. 4. Graphical Representation of Tasks

Proof: It is straight forward when we interpret the two tasks using the petri net in Fig. 3.

Theorem 2. *The above dependencies hold in the DVE code of sequential composition*

Proof: It is straight forward when we look at the DVE code for sequential composition which can be found in section 4.

Parallel Composition. Compensable tasks that are composed using parallel composition are executed in parallel. If one of the parallel tasks or branches fails or aborts then the entire composed transaction will fail or abort, as a composed transaction cannot reach its goal if a sub-transaction fails. The petri net representation of parallel composition of two compensable tasks ϕ_{c_1} and ϕ_{c_2} ($\phi_{c_1} \mid\mid \phi_{c_2}$) is shown in Fig. 5. Furthermore, parallel composition requires that if one branch either fails or aborts then the other branch should be stopped to save time and resources. This is achieved by an internal mechanism called *forceful abort* (not shown in Fig. 5), which forcefully aborts a transaction and undos its partial effects. Details of forceful abort can be found from our website [16]. To sum up, when compensating, tasks which are composed in parallel are required to be compensate in parallel.

The related behavioural dependencies are formalized as: **i)** $(\phi_{c_1}, act) \leftrightarrow (\phi_{c_2}, act)$; **ii)** $(\phi_{c_1}, und) \leftrightarrow (\phi_{c_2}, und)$; **iii)** $(\phi_{c_1}, suc) \leftrightarrow (\phi_{c_2}, suc)$. Analogous to the dependencies of the sequential composition, these basic dependencies also

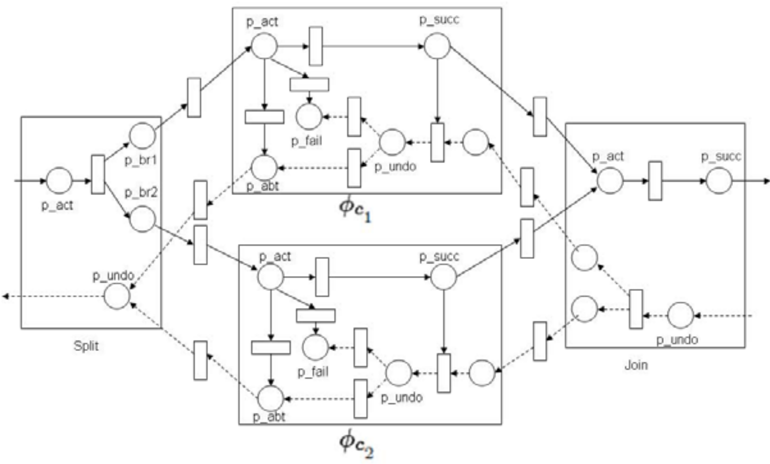


Fig. 5. Petrinet representation of parallel composition

holds in the petri net representation and in the DVE code of the parallel composition.

There can be more than two compensable tasks in parallel composition; for this composition all the branches will be activated and executed in parallel.

Alternative Forwarding. The alternative forwarding composition is used to decide between two or more equivalent tasks with the same goals. Alternative forwarding implies a preference between the tasks, and it does not execute all branches in parallel. Therefore if, for example, the alternative forwarding composition is used to buy air tickets, one airline may be preferred to the other and an order is first placed to the preferred airline. The other airline will be used to place an order only if the first order fails.

In Fig. 4, we can see the two tasks ϕ_{c_1} and ϕ_{c_2} which are composed by alternative forwarding. It represents the formula $\phi_{c_1} \rightsquigarrow \phi_{c_2}$. In this composition, task ϕ_{c_1} has higher priority and it will be executed first. Task ϕ_{c_2} will be activated only when task ϕ_{c_1} has been aborted or failed. In other words, ϕ_{c_1} runs first and ϕ_{c_2} is the backup of ϕ_{c_1}. The petri net representation of alternative forwarding composition of two compensable tasks ϕ_{c_1} and ϕ_{c_2} is shown in Fig. 6.

The basic dependency is described by: $(\phi_{c_1}, abt) < (\phi_{c_2}, act)$. Analogous to the dependencies of the sequential composition, this basic dependency also holds in the petri net representation and in the DVE code of the alternative forwarding composition.

Descriptions of Internal Choice, Speculative Choice, Backward handling, Forward handling and Programmable Compensations can be found from our website [16].

Construction Principle 1. *Construction principles for the graphical representation of tasks are as follows:*

- *The operators* $[\bullet, ;, \unrhd, \rhd, \circledast]$ *are used to connect the operand tasks sequentially. Atomic uncompensable tasks are connected by a single forward flow. Atomic*

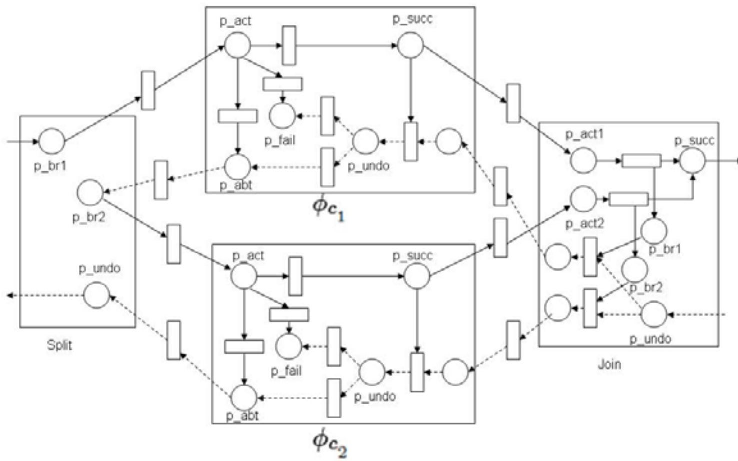


Fig. 6. Petrinet representation of alternative forwarding composition

compensable tasks are connected by two flows- one forward and one backward if they are connected by the sequential operator. Atomic uncompensable tasks and atomic compensable tasks are connected by a single forward flow;

– *The convention of ADEPT2 is followed in order to enable 'Poka-Yoke Work-flows' [17], which supports "correctness by construction". A pair of split and join routing tasks are used for tasks composed by* $\{\wedge, \vee, \times, ||, \sqcap, \otimes, \rightsquigarrow\}$. *Atomic uncompensable tasks are connected with split and join tasks by a single forward flow. Atomic compensable tasks are connected with split and join tasks by two flows (forward and backward). The operators and its corresponding split and join tasks are shown in Table 2;*
– *Two split and join tasks for the same operator can be merged to a single split task (or join task) combining the branches. By this arrangement tasks can be composed in more than two branches of a split/join pair.*
– *Every compensable task is covered by an exception handler (forward or backward).*

If these principles are followed, the resulting graph is said to be "correct by construction".

Table 2. Operators and their associated split and join tasks

Operator	Split task	Join task
$\wedge$	AND-split	AND-join
$\vee$	OR-split	OR-join
$\times$	XOR-split	XOR-join
$\|\|$	PAR-split	PAR-join
$\sqcap$	INT-split	INT-join
$\otimes$	SPEC-split	SPEC-join
$\rightsquigarrow$	ALT-split	ALT-join

3.3 Analysis

We adapt the definition of soundness for CWF-net from [13]. Informally, the soundness of CWF-net require that for any case, the underlying coloured petri net will terminate eventually, and at the moment it terminates, there is a token in the output condition and all other places are empty. Formally, the soundness of CWF-net is defined as follows:

Definition 8. *A CWF-net $C_N = (i, o, T, T_c, F)$ is sound (or structurally sound) iff, considering the underlying petri net:*

1. *For every state (marking) M reachable from the initial state M_i, there exists a firing sequence leading from M to the final state M_f, where M_i indicates that there is a token in the input condition and all other places are empty and M_f indicates that there is a token in the output condition and all other places are empty. Formally: $\forall M(M_i \rightarrow^* M) \Rightarrow (M \rightarrow^* M_f)$;*
2. *M_f is the only state reachable from M_i with at least one token in the output condition. Formally: $\forall_M (M_i \rightarrow^* M \wedge M \geq M_f) \Rightarrow (M = M_f)$;*
3. *There are no dead transitions in C_N. Formally: $\forall_{t \in T}, \exists_{M,M'} M_i \rightarrow^* M \rightarrow^t M'$.*

We have the following theorem:

Theorem 3. *If a CWF-net is correct by construction, it is sound.*

Proof: Let C_N be a CWF-net which consists of some uncompensable and compensable tasks.

- *Case 1,* C_N consists of only one atomic task (t): as t is connected to the input condition and the output condition. t will be activated by the input condition and will continue the forward flow to the output condition, the flow will terminate. Therefore C_N is sound.
- *Case 2,* C_N consists of only atomic uncompensable tasks composed by operators $\{\wedge, \vee, \times, \bullet\}$: according to the construction principle, every split task must have a same type of join task. This pair of split and join tasks provides a safe routing of the forward flow; all the tasks of the workflow are on a path from the input condition to the output condition, which ensures that there is no dead task in the workflow and the flow always terminates. Therefore C_N is sound.
- *Case 3,* C_N includes some atomic uncompensable tasks and atomic compensable tasks: First let us consider that C_N has one atomic compensable task (t_c). t_c is activated by some atomic task or the input condition. If t_c is successful during the execution, it will activate the next task (or the output condition) by continuing the forward flow. If t_c is aborted, it will start the compensation flow. As this is the only compensable task (it is the initial task itself), the compensation flow is connected to the output condition. It is easy to see that if t_c is aborted, the flow also terminates. If t_c fails during execution, the error handler will be in effect and will either terminate the flow to the output condition (backward handler) or continue the forward

flow (forward handler). Therefore C_N is sound. Now let us consider there is more than one atomic compensable task in C_N. For every compensable task there is an initial subtask and the compensation flow of the initial subtask is connected to the output condition. If the compensable tasks do not fail, they will continue the forward flow until the output condition is reached. If the composition of compensable tasks is aborted, the compensation flow will reach the initial subtask, which will direct the compensation flow to the output condition. Therefore C_N is sound.

4 Verification

Once a workflow is designed with compensable tasks, its properties can be verified by model checkers such as SPIN, SMV or DiVinE. Modeling a workflow with the input language of a model checker is tedious and error-prone. Leyla et al. [19] translated a number of established workflow patterns into DVE, the input language of DiVinE model checker. Based on this translation, we proposed an automatic translator which translates a graphical model constructed using the YAWL editor to DVE, greatly reducing the overall effort for model checking (see [20] and the extension involving time in [23]).

DiVinE is a distributed explicit-state *Linear Temporal Logic* (LTL) model checker based on the automata-theoretic approach. In this approach, the system properties are specified as LTL formulas. LTL is a type of temporal logic which, in addition to classical logical operators, uses the temporal operators such as: always ($\Box$), eventually ($\Diamond$), until ($\sqcup$), and next time ($\bigcirc$) [15]. In the automata-theoretic approach, the system model and the property formula are each associated to an automaton. The model checking problem is reduced to detecting an accepting cycle in the product of the system model automaton and the negation of the property automaton. DiVinE provides several model checking algorithms which efficiently employ the computational power of distributed clusters. DVE is sufficiently expressive to model general problems.

In [1] two types of verification were proposed for the verification of compensable transactions: i) Acceptable Termination States (ATS), ii) Temporal Verification. However they cannot verify the compensable transactions with the whole workflow, and cannot handle the state space explosion problem. Our approach can verify the workflow with the compensable tasks. As we are using a distributed model checker, we can handle the large state space of a large model. Since LTL specification formulas can be used to verify the temporal specification of the workflow it is possible to check the Acceptable Termination States (ATS). Behavioural dependencies and requirement specifications both can be checked using this approach.

In the DVE translation, workflow processes, subprocesses and activities are mapped to DVE processes and control flows are managed using DVE variables, guards and effects. We give one example below.

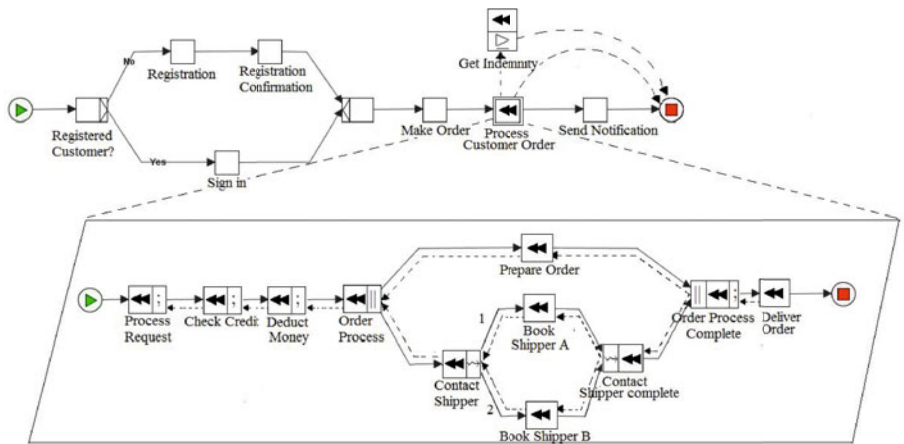


Fig. 7. Order processing workflow

Sequential Composition. Let A and B be two compensable tasks composed by the sequential operator (;). Each task corresponds to a process in DVE. When process A is activated, it can transit to `successful`, `aborted` or `failed`. B is activated only if A transits to `successful`. Once activated, B can also transit to `successful`, `aborted` or `failed`. If B is `aborted`, it sets positive value to the variable named B_abt (represents a token in its place p_abt).

```
int A_SUC = 0;
int A_ABT = 0;
. . . .
process A{
. . . .
tr -> tr { guard Start_SUC>0;effect Start_SUC=Start_SUC -1,A_SUC=A_SUC +1;},
tr -> tr { guard Start_SUC>0;effect Start_SUC=Start_SUC -1,A_ABT=A_ABT +1;},
tr -> tr { guard Start_SUC>0;effect Start_SUC=Start_SUC -1,A_FAIL= A_FAIL +1;},
tr -> tr { guard B_ABT > 0 ; effect B_ABT = B_ABT - 1  , A_ABT = A_ABT +1;},
tr -> tr { guard B_ABT > 0 ; effect B_ABT = B_ABT - 1 , A_FAIL = A_FAIL +1;};}
process B{
. . . .
tr -> tr { guard A_SUC > 0; effect A_SUC = A_SUC - 1, B_SUC = B_SUC + 1;},
tr -> tr { guard A_SUC > 0; effect A_SUC = A_SUC - 1, B_ABT = B_ABT + 1;},
tr -> tr { guard A_SUC > 0; effect A_SUC = A_SUC - 1, B_FAIL = B_FAIL + 1;};}
```

5 Case Study

In this section we provide a case study adapted from [1]. A customer's order processing workflow is described here with compensable transactions. Only a registered customer can make an order to the company. If the customer is not registered, s/he performs the registration and then signs in to the system. Once the order is made, the system starts processing the order. The workflow is presented in Fig. 7. In the workflow *Process Customer Order* the process is a nested

compensable task which has been decomposed into a CWF$_t$-net . The workflow provides lots of flexibility and exception handling through which the system can compensate in exceptional scenario. The *Prepare Order* and *Contact Shipper* are processed in parallel as these are two time consuming tasks. However if either *Prepare Order* or *Contact Shipper* aborts, the other will be aborted immediately. In this example, the seller has two shippers (A and B). *Shipper A* is cheaper but hard to book whereas *Shipper B* is more expensive but is always available. To save money, *Shipper A* is preferred; *Shipper B* is booked only when *Shipper A* is unavailable. The selected shipper is responsible for delivering these items. Note that if the customer cancels the order during processing, the compensation for completed parts will be activated. When the compensation cannot properly undo the partial effects, the seller would ask for extra indemnities from the customer which is transacted by backward handling. Here the backward handler is composed with a nested task which will be activated if any of the compensable tasks fails inside the CWF$_t$-net.

5.1 Verification Results

We have developed an editor for compensable transaction which includes a translator that can translate a CWF-net model to DiVinE model checker. The process is fully automated and the tool is available in our website [16].

Reachability: The reachability result shows that there is no deadlock state and no error state in the system. There are a total of 27620 states and 56089 transitions in experiment. We verified five properties, due to space limits, we only give details for *Prop1: Customer cannot make order without sign in or registration.*

To verify this property we define a global integer variable in DVE named `customer_sign_in`. The initial value of the variable is set to 0. The value is changed to 1 when the customer is signed in, means when the process *Sign_in* is in successful state. We define three atomic propositions in the following way:

```
#define order_made ( Make_Order_SUC == 1 )
#define sign_in ( customer_sign_in == 1 )
#define registration ( customer_registration == 1 )
```

The LTL specification using these propositions for the property is:

```
G F !( !( sign_in || registration) && order_made )
```

Other properties and the corresponding LTL formulas:

- *Prop2: Shipper A and B will not be contacted at the same time.*
  `G F !( shipper_a_is_successful && shipper_b_is_successful )`
- *Prop3: If Prepare Order is aborted, the Order Process task is compensated.*
  `G( prepare_order_aborted → F( order_process_compensated ) )`
- *Prop4: Get Indemnity is activated if Book Shipper A fails to compensate.*
  `G ( shipper_a_failed → F( get_indemnity_for_deduct_money ) )`
- *Prop5: Any order is eventually delivered or compensated if no task fails.*
  `G ( order_made → F ( order_delivered || order_compensated ) )`

Table 3. Verification Results for the DiVinE model checker

Property	Acc Cycle	States	Memory (MB)	Time (s)
Prop1	No	27623	166.891	0.388079
Prop2	No	27620	166.891	0.377525
Prop3	No	28176	167.012	0.370857
Prop4	No	27626	166.895	0.35127
Prop5	No	42108	170.023	0.623221

6 Conclusion and Future Work

Over the last decade, there has been increasing recognition that modeling languages should be more expressive and provide comprehensive support for the control-flow, data, resource and exception handling perspectives. In this paper we have shown how a workflow can be better represented by the composition of compensable tasks. The idea of workflow with compensable task will add a new dimension of flexibility and exception handling.

This research is part of an ambitious research and development project, Building Decision-support through Dynamic Workflow Systems for Health Care [21] in a collaboration with Guysborough Antigonish Strait Health Authority (GASHA) and technology industrial partner. Real world workflow processes can be highly dynamic and complex in a health care setting. To manage the ad hoc activities efficiently, a flexible workflow system with better exception handling mechanism must be designed. Hao and MacCaull [22] developed several new Explicit-time Description Methods (EDM), which enable general model checkers like DiVinE to verify real-time models. Based on them, we are extending CWML to describe real time information in workflow models. More importantly, our group is developing an innovative workflow modeling framework named *NOVA Workflow* that is compeNsable, Ontology-driven, Verifiable and Adaptive.

Acknowledgment

This research is sponsored by Natural Sciences and Engineering Research Council of Canada, by an Atlantic Computational Excellence Network (ACEnet) Post Doctoral Research Fellowship and by the Atlantic Canada Opportunities Agency. The computational facilities are provided by ACEnet.

References

1. Li, J., Zhu, H., He, J.: Specifying and verifying web transactions. In: Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K. (eds.) FORTE 2008. LNCS, vol. 5048, pp. 149–168. Springer, Heidelberg (2008)
2. Garcia-Molina, H., Salem, K.: Sagas. SIGMOD Rec. 16(3), 249–259 (1987)
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, pp. 133–150. Springer, Heidelberg (2005)

4. Bruni, R., Butler, M., Ferreira, C., Hoare, T., Melgratti, H., Montanari, U.: Comparing Two Approaches to Compensable Flow Composition. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 383–397. Springer, Heidelberg (2005)
5. He, J.: Compensable programs. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 349–363. Springer, Heidelberg (2007)
6. He, J.: Modelling coordination and compensation. In: Leveraging Applications of Formal Methods, Verification and Validation, vol. 17, pp. 15–36 (2009)
7. Li, J., Zhu, H., He, J.: Algebraic semantics for compensable transactions. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 306–321. Springer, Heidelberg (2007)
8. Li, J., Zhu, H., Pu, G., He, J.: Looking into compensable transactions. In: The 31st IEEE Software Engineering Workshop, pp. 154–166. IEEE CS press, Los Alamitos (2007)
9. Murata, T.: Petri nets: properties, analysis, and applications. Proc. IEEE 77(4), 541–580 (1989)
10. Narahari, Y., Viswanadham, N.: On the Invariants of Coloured Petri Nets. In: Rozenberg, G. (ed.) APN 1985. LNCS, vol. 222, pp. 330–345. Springer, Heidelberg (1986)
11. van der Aalst, W.M.P., ter Hofstede, A.: YAWL: Yet another workflow language. Inf. Syst. 30(4), 245–275 (2005)
12. Dadam, P., Reichert, M., Rinderle, S., et al.: ADEPT2 - Next Generation Process Management Technology. Heidelberger Innovationsforum, Heidelberg (April 2007)
13. Van der Aalst, W.M.P., Van Hee, K.: Workflow Management: Models, Methods and Systems. The MIT Press, Cambridge (2002)
14. DiVinE project, http://divine.fi.muni.cz/ (last accessed on August 2010)
15. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
16. Center for Logic and Information, St. Francis Xavier University, http://logic.stfx.ca/ (last accessed on August 2010)
17. Reichert, M., Dadam, P., Rinderle-Ma, S., et al.: Enabling Poka-Yoke Workflows with the AristaFlow BPM Suite. In: Dayal, U., Eder, J., Koehler, J., Reijers, H.A. (eds.) Business Process Management. LNCS, vol. 5701. Springer, Heidelberg (2009)
18. Barkaoui, K., Ben Ayed, R., Sbai, Z.: Workflow Soundness Verification based on Structure Theory of Petri Nets. International Journal of Computing and Information Sciences 5(1), 51–61 (2007)
19. Leyla, N., Mashiyat, A., Wang, H., MacCaull, W.: Workflow Verification with DiVinE. In: The 8th International Workshop on Parallel and Distributed Methods in verification, PDMC 2009 (2009) (work in progress report)
20. Rabbi, F., Wang, H., MacCaull, W.: YAWL2DVE: An automated translator for workflow verification. In: The 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010), pp. 53–59. IEEE CS press, Los Alamitos (2010)
21. Miller, K., MacCaull, W.: Toward Web-based Careflow Management Systems. Journal of Emerging Technologies in Web Intelligence (JETWI) Special Issue E-health Interoperability 1(2009), 137–145 (2009)
22. Wang, H., MacCaull, W.: An Efficient Explicit-time Description Method for Timed Model Checking. In: The 8th International Workshop on Parallel and Distributed Methods in verifiCation 2009 (PDMC 2009). EPTCS, vol. 14, pp. 77–91 (2009)
23. Mashiyat, A., Rabbi, F., Wang, H., MacCaull, W.: An Automated Translator for Model Checking Time Constrained Workflow Systems. In: FMICS 2010. LNCS, vol. 6371, pp. 99–114. Springer, Heidelberg (2010)

Automatically Testing Web Services Choreography with Assertions

Lei Zhou[1], Jing Ping[1], Hao Xiao[1],
Zheng Wang[1], Geguang Pu[1], and Zuohua Ding[2]

[1] Shanghai Key Laboratory of Trustworthy Computing,
Software Engineering Institute, East China Normal University,
Shanghai, 200062, China
[2] Center of Math Computing and Software Engineering,
Institute of Science, Zhejiang Sci-Tech University,
Hangzhou, Zhejiang, 310018, China
xiaohao@sei.ecnu.edu.cn, ray-zhou@hotmail.com,
pingjing1988@gmail.com, wangzheng@sei.ecnu.edu.cn,
ggpu@sei.ecnu.edu.cn, zuohuading@hotmail.com

Abstract. Web Service Choreography Description Language gives a global view on the collaborations among a collection of services involving multiple participants or organizations. Since WS-CDL is aimed at a design specification for service composition, there are few approaches to be proposed to test WS-CDL programs. In this paper, we present an approach to testing WS-CDL programs automatically. The dynamic symbolic execution technique is applied to generate test inputs and assertions are treated as the test oracles. Moreover, a simulation engine for WS-CDL is used to perform the execution of WS-CDL programs during the process of symbolic execution. At the end of each execution, the path conditions collected by symbolic execution are put into a SMT solver to generate new input data that will guide the next simulation. Meanwhile, the SMT solver is applied to decide whether the assertion predicates can be satisfied under current path conditions for all test data which improves the quality of testing further.

Keywords: Web Services, WS-CDL, Automatic Testing, Choreography, Symbolic Execution.

1 Introduction

The Web Services paradigm promises to carry on dynamic and flexible interoperability of highly heterogeneous and distributed platforms. SOAP [20], WSDL [21] and UDDI [12] specify the fundamental standards and others proceeded by WS-(WS-Policy, WS-Addressing, WS-Security etc. [22]) allow applications to interact with each other to form a loosely coupled platform-independent model.

The design and implementation of Web Services-based systems require the combination of different distributed services together to achieve the business goal. Thus, it is essential to use the logic operators on Web Services to compose and analyze the complex behaviors of the system. SOA community tries

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 138–154, 2010.
© Springer-Verlag Berlin Heidelberg 2010

to meet this demand by defining Web Services compositional languages such as WSFL [24], BPML [3], WS-BPEL [4], WS-CDL [23], and WSCI [25] etc. There are two representative models for Web Services composition. One is orchestration model, and the other is choreography model. The orchestration model gives a local view from one business part to handle interactions in which a given service can perform its internal activities as well as communicate with other services.

On the other hand, the choreography model gives a global view on the collaboration among a collection of services involving multiple different organizations or independent processes. Web Services Choreography Description Language (WS-CDL for short) is a W3C candidate recommendation for Web Services choreograph.

Fig. 1 depicts the structure the WS-CDL program. The root choreography is the entrance of the WS-CDL program and is defined uniquely. The activity notation defines specific logic for each choreography. In other words, both the entrance and the behavior of the choreographies are fixed.

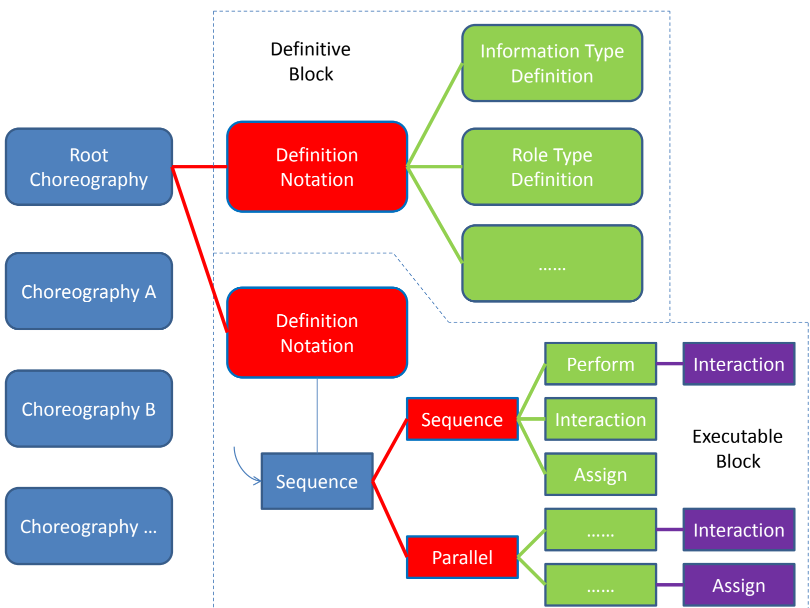


Fig. 1. WS-CDL Program Structure

The analysis of WS-CDL programs is important for the development of systems. If bugs can be removed as early as possible, especially at the design phase, the quality of systems can be improved while costs can be saved at the deployment phase. As its intuitive name indicates, WS-CDL is not an executable language, which makes it difficult to test, but testing for web service choreographies is an effective mechanism to ensure the qualities of system designs.

In this paper, we present a new approach to automatically testing WS-CDL programs. The basic idea is to automate test data generation based on the program structure according to the path coverage criteria and then use the generated data to execute the program automatically. The dynamic symbolic execution technique [17,7] is adopted to automate test data generation which guides the execution of programs. An assertion statement which is a language extension to the standard WS-CDL specification is developed to express the intention of the program by designers. To facilitate the automated testing of WS-CDL, we developed a simulation engine [27] for WS-CDL programs. However, one difficulty in testing WS-CDL is the parallel structure since the same test data may lead to different behaviors of the concurrent program. To handle this issue, we propose a practical method to test programs with parallel structures.

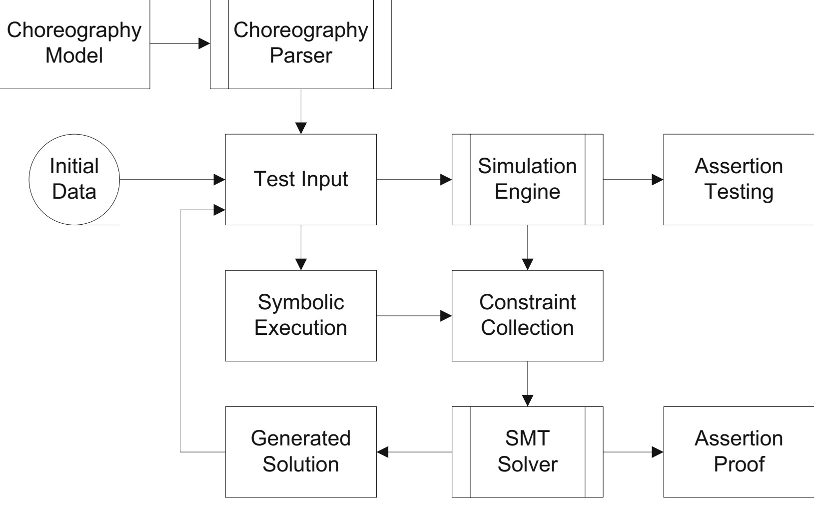


Fig. 2. Approach Outline

Fig. 2 demonstrates our approach. The WS-CDL program is firstly processed by the parser which is developed by ourselves. Since WS-CDL has an entrance unit, a random value is put into this unit to drive the choreography initially. Then, the simulation engine is performed to simulate program behaviors and record the choreography state at the same time. Meanwhile, the symbolic values are analyzed at the end of every simulation by means of symbolic execution. The predicates appeared in path branches are collected which make up the path constraint of the current program path. With a SMT solver (Z3 [29] is employed in our implementation) which solves the path constraint by negating the last term of the current path predicate, we can get a solution for the involved variables. This solution is taken as new input for service choreographies to guide the next simulation. The testing process terminates when all the paths in choreographies are traversed by the simulation engine. The assertion is designed to test whether

the current program point satisfies some property. We use two approaches to test assertions. The first one is using the generated test data to test whether the assertion is satisfied. However, while the current test data make the assertion true, there may be other data which make the assertion false. To enhance the assertion testing, we use the symbolic execution with a constraint solver to achieve a proof that can decide whether the assertion is satisfied under the execution of current path. This technique is similar to the one adopted by the bounded model checking for software [2]. When the assertion is satisfied by all the paths that pass it, then the assertion is proved to be true for service choreographies.

This paper is organized as follows. Section 2 describes the related work. Section 3 presents the motivating examples for our approach. Section 4 presents the automated test data generation approach, and some technical details are discussed as well. Based on our approach, we implemented the tool for automated testing of service choreography with assertions, and experimental results are introduced in Section 5. Section 6 concludes our work.

2 Related Work

The analysis of the service choreography has been studied in recent years. For instance, Foster et al. [6] proposed a model-based approach to the analysis of obligations in Web Services choreographies by means of process algebra. Pu et al. [13] proposed an approach to validation and verification of the WS-CDL specification with static and dynamic properties. Monakova et al. [18] proposed an approach to translate both WS-BPEL process and business rules into logic predicates, and then use a constraint solver to decide whether the business rules is valid under the current process model. Their work is different from ours since we use a constraint solver to generate test data instead of verification of business process.

Testing of Web Services is crucial for the practice of engineering [5,9,1,11]. For instance, Bartolini et al. [1] introduced a WS-TAXI framework to combine the coverage of WS operations with data-driven test generation. L. Mei et al. [11] proposed a C-LTS model to represent choreography applications from the testing perspective to address the challenges such as mismatches in message content selection. Kathrin Kaschner and Niels Lohmann [28] suggested an approach to automate test case generation based on the open nets [26] model of public protocol of interacting services.Yan et al. [14] applied symbolic execution to test BPEL4WS programs. They translated the BPEL4WS programs to a XCFG and simplified some features of BPEL4WS such as scope, exception handling. Our basic idea is similar to theirs, while we use dynamic symbolic execution, which can reduce the state space and support web service interactions better.

Our approach for testing WS-CDL needs a simulation engine, which has been developed by ourselves [15]. This engine differs from [19], in which the execution of WS-CDL programs will invoke the real services. Our simulation engine is actually a WS-CDL interpreter, which can simulate the WS-CDL language definitions, the interfaces and data types defined in services. Based on this simulator, we can carry on symbolic executions of WS-CDL programs thoroughly by providing all the possible data returned from services invocations.

3 Motivating Examples

In this section, two examples are presented. The first one is to demonstrate the basic idea about automatic testing of WS-CDL program. The second one handles the WS-CDL program with assertions.

The travel example is depicted in Fig. 3. It is a simple example about booking flight tickets and hotels before travelling. The dynamic symbolic execution starts with assigning random values to the program variables that affect the path selection. Variables $flightAvailable$ and $roomAvailable$ in this example are assigned to integer 0. We denote them as a pair of values $< 0, 0 >$.

```
<choreography name="TravelAgentChor" root="true">
    <relationship type="Customer_TravelAgent"/>
    <variableDefinitions>
    </variableDefinitions>
    <sequence>
        <workunit name="TestFlightAvailable" guard="cdl:getVariable(flightAvailable, TravelAgentRole)>0">
            <sequence>
                <workunit name="TestRoomAvailable" guard="cdl:getVariable(roomAvailable, TravelAgentRole)>0">
                    <sequence>
                        <perform choreographyName="ReserveFlightsChor"/>
                        <perform choreographyName="ReserveRoomsChor"/>
                    </sequence>
                </workunit>
            </sequence>
        </workunit>
    </sequence>
</choreography>
```

Fig. 3. Travel Example

The simulation engine of WS-CDL is firstly performed with initial input values. With the execution of the target program, the symbolic values of program variables are collected and stored with the concrete values. When the control flow arrives at the first *workunit*, the guard is not established since $[flightAvailable > 0]$ is not satisfied. As a result, the path constraint $\neg[availableFilgit > 0]$ is collected and the program terminates. To make the program traverse other paths, the negation of the last term in current path constraint is generated, as $[availableFilgit > 0]$. Then, a constraint solver (such as SMT solver) is called and a solution is obtained. Suppose integer 1 is returned for variable $flightAvailable$, and the new value pair $< 1, 0 >$ is generated which is put into the target program as a new test case. The engine is performed with the new test case, and constraint path $[flightAvailable > 0] \wedge [\neg(roomAvailable > 0)]$ is collected with the current execution. In this case, the first *workunit* is satisfied while the second *workunit* fails to be true. Thus, the negation of the last term $[\neg(roomAvailable > 0)]$ is obtained and the constraint solver generates the new solution for the constraint $[flightAvailable > 0] \wedge [roomAvailable > 0]$. Suppose the new solution is $< 1, 1 >$ which is regarded as a new data for the next execution. As a result, the two *workunit*s are all satisfied and the two sub-choreographies are performed afterwards. Thus, three test cases are generated automatically and cover all the program paths.

```
<choreography name="TravelAgentChor" root="true">
    <relationship type="Customer_TravelAgent"/>
    <variableDefinitions>
    </variableDefinitions>
    <sequence>
        <workunit
            name="TestFlightAvailable"
            guard="cdl:getVariable(flightAvailable, TravelAgentRole) &gt; cdl:getVariable(flightRequire, CustomerRole)">
            <sequence>
                <workunit
                    name="TestRoomAvailable"
                    guard="cdl:getVariable(roomAvailable, HotelRole) &gt; cdl:getVariable(roomRequire, CustomerRole)">
                    <sequence>
                        <parallel>
                            <workunit
                                name="OrderRoom"
                                repeat="cdl:getVariable(roomOrdered, CustomerRole) &lt; cdl:getVariable(roomRequire, CustomerRole)">
                                <sequence>
                                    <perform choreographyName="ReserveRoomsChor"/>
                                    <assign roleType="customerRole">
                                        <copy name="addOrderedRoom">
                                            <source expression="cdl:getVariable(roomOrdered, CustomerRole) + 1"/>
                                            <target variable="cdl:getVariable(roomOrdered, CustomerRole)"/>
                                        </copy></assign></sequence>
                            </workunit>
                            <workunit
                                name="OrderFlight"
                                repeat="cdl:getVariable(flightOrdered, CustomerRole) &lt; cdl:getVariable(flightRequire, CustomerRole)">
                                <sequence>
                                    <perform choreographyName="ReserveFlightsChor"/>
                                    <assign roleType="customerRole">
                                        <copy name="addOrderedFlight">
                                            <source expression="cdl:getVariable(flightOrdered, CustomerRole) + 1"/>
                                            <target variable="cdl:getVariable(flightOrdered, CustomerRole)"/>
                                        </copy></assign></sequence>
                            </workunit>
                        </parallel>
                    </sequence>
                </workunit>
            </sequence>
        </workunit>
        <assertion guard="cdl:getVariable(roomAvailable, HotelRole) &gt;0"/>
        <assertion guard="cdl:getVariable(flightAvailable, TravelAgentRole) &gt;0"/>
    </sequence>
    <assertion guard="cdl:getVariable(roomOrdered, HotelRole) = cdl:getVariable(roomRequire, CustomerRole)"/>
    <assertion guard="cdl:getVariable(flightAvailable, TravelAgentRole) = cdl:getVariable(flightRequire, CustomerRole)"/>
</choreography>
```

Fig. 4. Travel Example with Assertions

In order to express the expected behaviors of service choreographies, we design the assertion statements for WS-CDL, and the details are discussed in Section 4. Here, one example is presented to show how the assertion mechanism works. We add two artificial assertions into the previous example and it is depicted in Fig. 4. The first assertion says that variable $flightAvailable$ is less than 3 while the second assertion decides that $roomVailable$ is more than 0. When the last test case $< 1, 1 >$ is generated and executed, it passes both of the assertions. Though the two assertions are both passed, but the first assertion may be failed while the second assertion is always true. For instance, if the generated data is $< 4, 1 >$ that still satisfies the two *workunit*s, while it cannot pass the first assertion. By symbolic execution, assertions are also collected with the path constraints. For instance, when the program executes with the test data $< 1, 1 >$ and terminates, we obtain the following predicate formulas:

$$[flightAvailable > 0] \wedge [(roomAvailable > 0)] \rightarrow [flightAvailabe < 3]$$

$$[flightAvailable > 0] \wedge [(roomAvailable > 0)] \rightarrow [flightAvailabe > 0]$$

When the two formulas are put into the SMT solver, we can obtain that there exists a solution that makes the first formula false while the second is always true. Obviously, this technique can enhance the testing and the details are introduced in Section 4.

4 Automated Test Data Generation

The simulation engine can simulate one single execution of choreography for one test input. To automatically test choreographies, it is important to automate the test data generation which satisfies some coverage criteria. After test data have been generated, the simulation engine can perform WS-CDL program with those generated test inputs one after another, thus the testing process can be fully automated. The coverage criteria we adopt is the path coverage, and each test case will exercise one path in the WS-CDL program. In the following we will explain how our approach works.

4.1 Control Flow Graph (CFG)

To perform symbolic execution, both data and control flow information should be collected from WS-CDL program under test. The Control Flow Graph [10] is used to represent the control flow information for WS-CDL program. We formally introduce CFG below. A choreography can be modeled as its corresponding control flow graph.

Definition 1. $CFG = (V, V_B, \mathcal{L}, l_0, op, E)$, *where*

- *a set of variables V, and $V_B \subseteq V$ denotes binding variables, which plays the role as input and output variables.*
- *a set of control labels $\mathcal{L}$ and $l_0 \in \mathcal{L}$ is the start label*
- *a mapping op from a label $l \in \mathcal{L}$ to the following basic instruments:*
  1. stop*: termination instrument.*
  2. $x := e$*: assignment instrument, where $x \in V$ and e is arithmetic expression over V.*
  3. if e then l_1 else l_2*: conditional instrument, where e is arithmetic expression over V and $l_1 \in \mathcal{L}$, $l_2 \in \mathcal{L}$.*
  4. throw$(en,\ l_1)$*: the instrument that can throw an exception, where en is the exception name and l_1 is the label of the exceptionBlock of current choreography.*
  5. catch case $en_1 : l_1$, case $en_2 : l_2$, $\cdots$, case $en_n : l_n$, else $: l_0$*: the instrument that handles exceptions or passes exceptions to its immediately enclosing choreography, where $l_1, \cdots, l_n$ is an list of exception names which it take interests in and l_0 is the immediately enclosing choreography's exceptionBlock, l_n may be "default" which means it is interested in all exceptions.*
  6. finalize(fn, l_1)*: finalize instrument where fn is the name of finalizerBlock construct in its immediately enclosed choreography and l_1 is the finalizerBlock's label.*
  7. perform$(ch, id, \overrightarrow{b})$*: perform instrument, where ch is the name of performed choreography, id is the instance id of the performed choreography and $\overrightarrow{b}$ is a mapping from variables in the performing to variables in the performed choreography, domain$(\overrightarrow{b}) \subseteq V$.*

– *a set of directed edges* $E \subseteq \mathcal{L} \times \mathcal{L}$ *is defined as follows:*
  1. *if* $l \in \mathcal{L}$ *and* $op(l)$ *is an assignment instrument or a perform instrument, then there is exactly one* $l' \in \mathcal{L}$ *with* $(l, l') \in E$
  2. *if* $l \in \mathcal{L}$ *and* $op(l)$ *is an conditional instrument* if e then l_1 else l_2, *then* $(l, l_1) \in E$ *and* $(l, l_2) \in E$.

Since there are some WS-CDL constructs which are not stated above, but they can be translated into above CFGs. The transformation follows the rules defined below:

– assign, exchange, record elements are modeled with assignment.
– workunit is equal to if *guard* then $\{activity;$ while $repeat \wedge guard$ do $activity\}$, and while b do *activity* statement is flattened to compound if statements with a fixed maximum repetition number of if embedded in the then branch (if b then$\{activity;$ if b then$\{$ $activity; \cdots\}\}\}$).
– The perform element in WS-CDL has an attribute *block* which specifies that when set to true, the performed choreography must be processed with the performing one concurrently.
– *causeException* attribute can be modeled with throw and timeout can be translated to if e then throw(timeout, l_1).
– activities parallel and perform with attribute *block* set to false are both required to run concurrently, there may be data racing in concurrent programs. Parallel activities are first detected whether there are shared variables by concurrent programs, if there is no data racing, we interleave the activities in the order as they are defined; if there is data racing, enumerate all the interleaves and the simulation is controlled by a scheduler to make sure these activities run as the deterministic order.
– choice elements allow for nondeterministic choices between different alternatives. In order to cover all the paths, every alternative will be scheduled to run once. So there may be more than one path for one single test input.

Based on the above definition, the labels of a CFG correspond to choreography elements with associated instruments, and edges correspond to control flow from one instrument to the next. We assume that there is exactly one label l_{stop} in a CFG with $op(l_{stop}) = stop$, which can be guaranteed by our CFG construction step. A path is a sequence of labels $l_1, l_2, ..., l_n$ in the CFG, where l_1 is l_0 and l_n is l_{stop}. A label $l \in \mathcal{L}$ is reachable if there is a path $l_0, ..., l$ in the CFG.

4.2 Dynamic Symbolic Execution

Dynamic symbolic execution uses the following idea: the program under test is executed by supplying concrete values for input variables. During the concrete execution, the symbolic program state is updated with symbolic expressions over variables and the path constraints are collected at every conditional branch along the path.

Semantics. We illustrated the denotational semantic for WS-CDL in [16]. Here we present the operational semantics with a memory model that maps variables in V to values for comprehension of symbolic execution. For a concrete memory M, we use the notation $M[x \rightarrow v]$ to denote the mapping from variable x to its value v. For expression e, its value is denoted by $M(e)$ where every variable x occurring in e is replaced by value $M(x)$. We use μ to denote the symbolic memory. During symbolic execution, each instrument updates the symbolic memory and the control label. Suppose that the current program symbolic state is $< l, \mu, s >$, where l is the current label, and s is a mapping from a choreography name to a symbolic memory μ, which is used to manipulate the finalizer. The updating rules are defined as follows. We omit describing the unchanged elements in the configuration.

1. if $op(l)$ is $x := e$, then $l' = l_1$ and $\mu' = \mu[x \rightarrow \mu(e)]$, where $(l, l_1) \in E$.
2. if $op(l)$ is $\textsf{if}\ e\ \textsf{then}\ l_1\ \textsf{else}\ l_2$ and $M(e) \neq 0$, then $l' = l_1$.
3. if $op(l)$ is $\textsf{if}\ e\ \textsf{then}\ l_1\ \textsf{else}\ l_2$ and $M(e) = 0$, then $l' = l_2$.
4. if $op(l)$ is $\textsf{throw}(en, l_1)$ and $M(exception) = 0$, then $l' = l_1$ and $M'[exception \rightarrow 1]$ and $M'[EN \rightarrow en]$.
5. if $op(l)$ is $\textsf{catch case}\ en_1 : l_1,\ \textsf{case}\ en_2 : l_2,\ \cdots, \textsf{case}\ en_n : l_n, \textsf{else} : l_0$ and $M(exception) = 1$ and $M(EN) = en_k$ where $1 \leq k \leq n$ then $l' = l_k$,$M' = M[EN \rightarrow null; exception \rightarrow 0]$.
6. if $op(l)$ is $\textsf{catch case}\ en_1 : l_1,\ \textsf{case}\ en_2 : l_2,\ \cdots, \textsf{case}\ en_n : l_n, \textsf{else} : l_0$ and $M(exception) = 1$ and $l_n = default$ then $l' = l_n$,$M' = M[EN \rightarrow null; exception \rightarrow 0]$.
7. if $op(l)$ is $\textsf{catch case}\ en_1 : l_1,\ \textsf{case}\ en_2 : l_2,\ \cdots, \textsf{case}\ en_n : l_n, \textsf{else} : l_0$ and $M(exception) = 1$ and $l_n \neq default$ and $\forall 1 \leq k \leq n \cdot M(EN) \neq l_k$ then $l' = l_0$,$M' = M$.
8. if $op(l)$ is $\textsf{finalize}(ch, f)$ then $l = finalizer(ch, f)$ and $\mu' = get(s, ch)$.
9. if $op(l)$ is $\textsf{stop}$
   - if l is in the CFG constructed from the root choreography, then the symbolic execution terminates.
   - on the other case, assume that this choreography ch is performed by the instrument $\textsf{perform}(ch, \overrightarrow{b})$, and the performer has the symbolic state $\langle l_p, \mu_p, s_p \rangle$, then $l' = l_1$ and $\mu' = \forall x \in domain(\overrightarrow{b}) \cdot (\mu_p[x \rightarrow \mu(\overrightarrow{b}(x))])$, where $(l, l_1) \in E$, and E is in the CFG constructed from the choreography performer.
10. if $op(l)$ is $\overrightarrow{r} := \textsf{perform}(ch, \overrightarrow{e})$, the current symbolic state $< l, \mu, s >$ is preserved and $l' = l_0$, $\mu' = \begin{cases} \mu'[x \rightarrow \mu(e)], x \in V_B, e \in \overrightarrow{e} \\ \mu'[x \rightarrow x], x \in V/V_B \end{cases}$, $s' = put(s, \langle ch, \mu \rangle)$, where l_0 and X_0 are in the CFG of the performed choreograph.

Rules 1, 2 and 3 are trivial, and rules 4, 5 and 6 deal with the exception block where there are 3 cases: case 1 donates that when the thrown exception matches one of the defined exceptions then the activity is performed within defined within

that exception workunit; case 2 donates that when there is a default handler then uncaught exceptions will be handled by it; case 3 donates that when there is no default exception and no matched handler then the exception is thrown out to its enclosing choreography's exception block. Rule 8 describes the finalization mechanism. The state is restored and then the specified $finalizerBlock$ is executed. Rules 9 and 10 handle the **perform** instrument. Rule 9 denotes that when a choreography ch_2 terminates, if it is not the root choreography, the simulator should restore the context for the choreography ch_1 performing ch_2 and update the variable state of ch_1 if there exists any variable binding between ch_1 and ch_2. Rule 10 says that when a choreography ch_1 performs the other choreography ch_2, the simulator must preserve the context for ch_1 and initialize ch_2 with the input from ch_2.

Symbolic Execution Algorithm. The symbolic execution algorithm is illustrated in Fig. 5. This algorithm is performed on the CFG, using both concrete memory M and symbolic memory μ. The path constraint $\mathfrak{P}$ collects predicates over symbolic values of variables along the execution path.

Symbolic execution starts at label l_0 in the CFG constructed from root choreography. In this case, the set of binding variables V_B is empty and variables in μ are initialized to itself while the concrete memory M is set to initial test inputs. If the choreography ch is performed by another choreography ch_0, then the binding variables in ch will be initialized by their corresponding variables in ch_0 (line 1).

For an assignment instrument, the algorithm updates both concrete and symbolic memories. And the current label is updated to be the next label in the CFG (line 5). For a conditional instrument, neither concrete nor symbolic memory is updated. The path constraint is conjunct with conditional predicate (b) or its negation ($\neg b$), depending on its evaluation on concrete memory. The current label is updated to l_1 if the predicate is evaluated non-zero or l_2 if zero (line 7).

We introduce a function $finalizer(ch, f)$ to specify the finalize structure. This function returns the label of entry node in the $finalizerBlock$ called by the finalize structure. The symbolic memory is restored from the record s (line 19).

To represent **perform** instrument, we introduce an operator $\oplus$ over two mappings, which is defined as follow:

$$(m_1 \oplus m_2)(x) \triangleq \begin{cases} m_1(x), x \notin domain(m_2) \\ m_2(x), x \in domain(m_2) \end{cases}$$

We first use recursion to execute the performed choreography symbolically, and then merge its effect with the performer. The path constraints collected from the performed choreography are attached to the current path constraints, and side-effect is merged to current memory (line 21).

The recursive procedure returns when the simulator hits a **stop** instrument. If the CFG is constructed from the root choreography, the execution terminates. Otherwise, the simulator must recover the context. An operator $\upharpoonright$ projecting a

Algorithm: SymbolicExecution(P, $\overrightarrow{e}$)
Inputs:
P is $CFG = (V, V_B, \mathcal{L}, l_0, op, E)$constructed from a choreography.
$\overrightarrow{e}$ is the set of input values.
Outputs:
$\mathfrak{P}$ is the path constraint.
μ_B is the mapping from binding variables to their symbolic values.
M_B is the mapping from binding variables to their concrete values.

1: $\mu = \begin{cases} \mu[x \to s(e)], x \in V_B, e \in \overrightarrow{e} \\ \mu[x \to x], x \in V/V_B \end{cases}$, $M = \begin{cases} \mu[x \to e], x \in V_0, e \in \overrightarrow{e} \\ \mu[x \to default], x \in V/V_0 \end{cases}$
2: Current label $l = l_0$, Path constraint $\mathfrak{P} = TRUE$
3: **while** $op(l) \neq$ stop **do**
4: **switch** $op(l)$
5: **case** $x := e$: $l' = l_1$,$M' = M[x \to M(e)]$, $\mu' = \mu[x \to \mu(e)]$, where $(l, l_1) \in E$
6: **case** if e then l_1 else l_2:
7: $\mathfrak{P} = \begin{cases} \mathfrak{P} \wedge \mu(e),\ M(e) \neq 0 \\ \mathfrak{P} \wedge \neg\mu(e),\ M(e) = 0 \end{cases}$, $l' = \begin{cases} l_1,\ M(e) \neq 0 \\ l_2,\ M(e) = 0 \end{cases}$
8: **case** throw(en, l_1):
9: $l' = \begin{cases} l_1,\ M(EN) = null \wedge M(exception) = 0 \\ l,\ M(EN) \neq null \vee M(exception) \neq 0 \end{cases}$,
10: $\mathfrak{P} = \begin{cases} \mathfrak{P} \wedge M(EN) = null \wedge M(exception) = 0,\ if \\ \qquad M(EN) = null \wedge M(exception) = 0 \\ \mathfrak{P},\ M(EN) \neq null \vee M(exception) \neq 0 \end{cases}$,
11: $M' = \begin{cases} M[exception \to 1 if EN \to en]\ if \\ \qquad M(EN) = null \wedge M(exception) = 0 \\ M if M(EN) \neq null \vee M(exception) \neq 0 \end{cases}$,
12: $\mu' = \begin{cases} \mu[EN \to en, exception \to 1],\ M(EN) = null \wedge M(exception) = 0 \\ \mu,\ M(EN) \neq null \vee M(exception) \neq 0 \end{cases}$
13: **case** catch case $en_1 : l_1$, case $en_2 : l_2$, $\cdots$,case $en_n : l_n$, else : l_0:
14: $l' = \begin{cases} l_k,\ M(exception) = 1 \wedge (M(EN) en_k \vee l_n = default) \\ l_0,\ M(exception) = 1 \wedge M(EN) \neq en_k \wedge l_n \neq default \end{cases}$,
15: $\mathfrak{P} = \begin{cases} \mathfrak{P} \wedge \mu(EN) \neq null \wedge \mu(exception) = 1,\ if \\ \qquad M(exception) = 1 \wedge (M(EN) = en_k \vee l_n = default) \\ \mathfrak{P},\ M(exception) = 1 \wedge M(EN) \neq en_k \wedge l_n \neq default \end{cases}$,
16: $M' = \begin{cases} M[EN \to null, exception \to 1],\ if \\ \qquad M(exception) = 1 \wedge (M(EN) = en_k \vee l_n = default) \\ M,\ M(exception) = 1 \wedge M(EN) \neq en_k \wedge l_n \neq default \end{cases}$,
17: $\mu' = \begin{cases} \mu[EN \to en, exception \to 1],\ if \\ \qquad M(exception) = 1 \wedge (M(EN) = en_k \vee l_n = default) \\ \mu,\ M(exception) = 1 \wedge M(EN) \neq en_k \wedge l_n \neq default \end{cases}$
18: **case** finalize(ch, f):
19: $\mu' = get(s, ch)$, $l' = finalizer(ch, f)$
20: **case** perform$(ch, \overrightarrow{b})$:
21: $(r_1, r_2, r_3) = SymbolicExecution(cfg(ch), \overrightarrow{b})$,
$\mathfrak{P} = \mathfrak{P} \wedge r_1$,$\mu = \mu \oplus r_2$, $M = M \oplus r_3$
22: **end switch**
23: **end while**
24: **if** P is constructed from a root choreography
25: **return** ($\mathfrak{P}$,ϕ, ϕ)
26: **else**
27: **return** ($\mathfrak{P}$,$\mu \restriction V_B$, $M \restriction V_B$)
28: **end if**

Fig. 5. Symbolic Execution Algorithm

mapping on a set of variables is introduced to represent the effect of variable binding (line 27):

$$(m \upharpoonright V)(x) \triangleq \begin{cases} m(x), x \in V \\ \perp, x \notin V \end{cases}$$

For an execution path, every assignment satisfying the path constraints $\mathfrak{P} = p_1, p_2, ..., p_n$ gives an instance of test case that guarantees the concrete execution will cover this path. If we negate some $p_i \in \mathfrak{P}$ and make conjunctions $p_1, ..., p_{i-1}, \neg p_i$, By solving the new path constraints through SMT solver, new test data will be generated under which the execution will cover the path constraints.

4.3 Assertion

In order to express the expected outcomes of correct program executions, we designed an assertion statement for WS-CDL to test whether an execution satisfies a custom intension. The assertion is a language extension to WS-CDL. And its XML schema is shown in Fig. 6.

```
<complexType name="tAssertion">
    <complexContent>
        <extension base="cdl:tExtensibleElements">
            <attribute name="guard" type="cdl:tBoolean-expr"></attribute>
        </extension>
    </complexContent>
</complexType>
```

Fig. 6. XML Schema of Assertion

Assertion statement acts like *workunit* statement without executing internal activity notation. As the simulator encounters an assertion, it first evaluates the guard expression with concrete values and then performs the assertion proof process in Fig. 2. The assertion proof process starts with substituting the variables in assertions with their symbolic values to get new assertion predicate (AP)and uses Z3 to prove or disprove that current path constraint (PC) implies the assertion predicate ($\mathtt{PC} \rightarrow \mathtt{AP}$).

5 Implementation and Experiments

In our previous work, we developed a WS-CDL simulator "CDLChecker" [27]. Based on this tool, we have implemented the symbolic execution framework of WS-CDL and developed the test input generation module for WS-CDL. There are two parts in this section. In the first sub-section, we give a brief introduction of our implementation. The effectiveness of our approach is shown through experiments in the second sub-section.

5.1 Tool Implementation

In this section, we present the implementation of our approach based on the "CDLChecker" tool. Figure 7 shows the architecture of the tool.

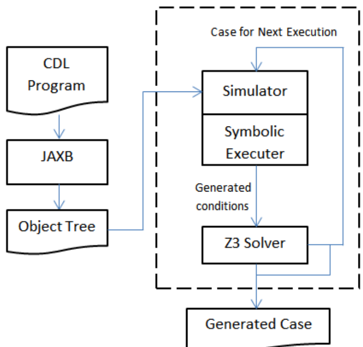


Fig. 7. Symbolic Execution Architecture of WS-CDL Checker

CDLChecker parses WS-CDL programs into Java object trees using JAXB [8] technique, an XML access mechanism for Java. The subsequent test and simulations are based on the Java object tree. Firstly, schema-derived classes are generated from WS-CDL schema, and WS-CDL programs are instantiated into object trees by these schema-derived classes. These objects are either definition structures or executable structures.

The simulator with symbolic execution embedded in takes the object tree as input to perform generation and collect path conditions. It transforms all definition structures into specific data structures and executes all executable structures when simulating each choreography. All integer variable instances within the entire choreography are assigned to 0 as initial inputs for the choreography. Assignment statements are executed both concretely and symbolically. In other words, both concrete values and symbolic values are recorded in *variabletable*. When executing *workunit* structures, guard conditions are evaluated by replacing variables with concrete values to determine the alternatives. The conditions are collected symbolically by replacing variables with their symbolic values and recording boolean results. After one pass of automatic simulation, a list of path conditions and their execution results are formed. Then we can negate the last path condition item and put it into Z3 [29] to get a new set of values which will guide the simulator to cover another path. Once all program paths have been covered, this generation procedure terminates.

Figure 8 presents a sample case to illustrate the tool. The program consists of two nested *workunit* elements to test whether specified variables satisfy some conditions. The log of the generation procedure is to show the outcomes, indicating that there are three possible paths within the program and three sets of test data corresponding to each path are provided.

```
<choreography name="TravelAgentChor" root="true">
  <relationship type="Customer_TravelAgent"/>
  <variableDefinitions>
  </variableDefinitions>
  <sequence>
    <workunit
      name="TestFlightSeatAvailablity" guard="cdl:getVariable(flightAvailable, TravelAgentRole) &gt; 0">
      <sequence>
        <workunit
          name="TestRoomAvailable" guard="cdl:getVariable(roomAvailable, TravelAgentRole)&gt; 0">
          <sequence>
            <perform choreographyName="ReserveFlightsChor"/>
            <perform choreographyName="ReserveRoomsChor"/>
          </sequence>
        </workunit></sequence></workunit></sequence>
</choreography>
```

Generation Start
  1st round...
    generated case:
    TravelAgentRole.flightAvailable = 1:
  2nd round...
    generated case:
    TravelAgentRole.flightAvailable = 1:
    TravelAgentRole.roomAvailable = 1:
  3rd round...
    Generation finished.

Generation Summary:
  Total 3 case(s) to 3 path(3)
    case 1: 0 for each variable
    case 2: TravelAgentRole.flightAvailable > 0
    case 3: TravelAgentRole.rootAvailable>0 $\wedge$ TravelAgentRole.flightAvailable>0

Fig. 8. Generated Case for A Sample Case

5.2 Preliminary Experiments

We tested five WS-CDL programs on the CDLChecker tool. The five programs cover most of the features in WS-CDL specification. We record the following statistics for each program in Table 1: time taken to run the program (the Time(ms) column), the number of path existed in the WS-CDL program (the #Existed Paths column), number of paths covered by the generated test inputs

Table 1. Experimental Results

NO	#Test Case	Time(ms)	#Existed Paths	#Covered Paths	#Assertions	#Violated Assertions
1	7	109	8	6	2	1
2	4	31	4	3	1	0
3	7	131	11	5	3	0
4	4	40	11	3	3	0
5	6	75	12	3	1	1

(the #Covered Paths column), number of asserts (the #Asserts) and number of violating asserts in testing phase(the #Violated Asserts).

The path coverage is less than 50% in the program no.3, no.4 and no5, and it is because the number of existed paths is counted from the CFG and the feasibility of the paths is not considered. Two assertion violations are detected when testing program no.1 and no.5. The segment of program no.1 in Fig. 9 shows how the assertion is violated.

Generate input:
flightsAvailable = -2

```
<?xml version="1.0" encoding="UTF-8"?>
<package name="travelAgentChoregraphy">
    ...
    <sequence>
        <choice>
            <workunit
                name="TestWU8"
                guard="cdl:getVariable('tns:flightsAvailable', 'tns:CustomerRole') &lt;= -2">
                <assertion guard="cdl:getVariable('tns:flightsAvailable', 'tns:CustomerRole') &lt;= -3" />
            </workunit>
            ...
        </choice>
    </sequence>
    ...
</package>
```

The guard of workunit TestWU8 is satisfied

The guard of assertion is violated.
Because flightsAvailable = -2, and the assertion requires flightsAvailable = -3

Fig. 9. Assertion Violation

6 Conclusions

In this paper, we present an approach to automatically generating test data for WS-CDL programs with assertion using symbolic execution technique, based on our previous work on simulation of WS-CDL programs. Two examples are presented to demonstrate the basic idea about automatic testing and a newly added assertion statement in WS-CDL program. We developed a choreography simulator to execute WS-CDL program in our previous work. On the basis of simulation, symbolic execution is applied to collect path conditions with detecting data racing in parallel situation. We use Z3 as a SMT solver to obtain the test data from the collected and processed path conditions. Moreover we defined an assertion statement for WS-CDL to check whether an execution of program satisfies the intentions of the designer. Based on our work, Web Services Choreography can be tested upon all possible paths, and unreachable paths can be detected as well.

Acknowledgement

Geguang PU is partially supported by 973 Project No.2005CB321904, and the Fundamental Research Funds for the Central Universities. Xiao HAO is partially

supported by 863 Project No.2009AA010313. Lei ZHOU is partially supported by NFSC No.90818024. Zheng WANG is partially supported by Shanghai Leading Academic Discipline Project No.B412. Zuohua DING is partially supported by NNSFC No.90818013.

References

1. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: WS-TAXI: A WSCDL-based Testing Tool for Web Services. In: ICST 2009, pp. 326–335 (2009)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. In: Advances in Computers, vol. 58, pp. 118–149 (2003)
3. Business Process Modeling Language (BPML), http://www.ebpml.org/bpml.htm
4. Business Process Execution Language for Web Services version 1.1 http://www.ibm.com/developerworks/library/specification/ws-bpel/
5. Chan, W.K., Cheung, S.C., Leung, K.R.P.H.: Towards a Metamorphic Testing Methodology for Service-oriented Software Applications. In: QSIC 2005, pp. 470–476 (2005)
6. Foster, H., Uchitel, S., Magee, J., Kramer, J.: Model-Based Analysis of Obligations in Web Service Choreography. In: AICT/ICIW 2006, p. 149 (2006)
7. Godefroid, P., Klarlund, N., Sen, K.: Dart: Directed Automated Random Testing. In: PLDI 2005, pp. 213–223 (2005)
8. Java Architecture for XML Binding (JAXB), https://jaxb.dev.java.net/
9. Li, Z., Sun, W., Jiang, Z.B., Zhang, X.: BPEL4WS Unit Testing: Framework and Implementation. In: ICWS 2005, pp. 103–110 (2005)
10. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (2004)
11. Mei, L., Chan, W.K., Tse, T.H.: Data Flow Testing of Service Choreography. In: ESEC/FSE 2009, pp. 151–160 (2009)
12. OASIS. Universal Description Discovery and Integration (2004), http://www.uddi.org/pubs/uddi_v3.htm
13. Pu, G., Shi, J., Wang, Z., Jin, L., Liu, J., He, J.: The Validaton and Verification of WSCDL. In: APSEC 2007, pp. 81–88 (2007)
14. Yan, J., Li, Z., Yuan, Y., Sun, W., Zhang, J.: BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach. In: ISSRE 2006, pp. 75–84 (2006)
15. Zhou, L., Zhang, H., Wang, T., Yang, C., Wang, Z., Sun, M., Pu, G.: Static Check of WS-CDL Documents. In: SOSE 2008, pp. 142–147 (2008)
16. Pu, G., Zhao, Y., Wang, Z., Feng, L., Zhu, H., He, J.: A Denotational Model for Web Services Choreography. In: Parashar, M., Aggarwal, S.K. (eds.) ICDCIT 2008. LNCS, vol. 5375, pp. 1–12. Springer, Heidelberg (2008)
17. Sen, K., Marinov, D., Agha, G.: Cute: A Concolic Unit Testing Engine for C. In: ESEC/FSE-13, pp. 263–272 (2005)
18. Monakova, G., Kopp, O., Leymann, F., Moser, S., Schafers, K.: Verifying Business Rules Using an SMT Solver for BPEL Processes. In: BPSC 2009, pp. 81–94 (2009)
19. Kang, Z., Wang, H., Hung, P.C.: WS-CDL+ for web service collaboration Information Systems Frontiers, vol. 9, pp. 375–389. Kluwer Academic Publishers, Dordrecht (2007)

20. W3C Note. Simple Object Access Protocol (SOAP) 1.1 (2000), http://www.w3.org/TR/soap
21. W3C Note. Web Service Definition Language (WSDL) 1.1 (2001), http://www.w3.org/TR/wsdl
22. Weerawarana, S., Curbera, F., Leymann, F., Storey, T., Ferguson, D.F.: Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, Englewood Cliffs (2005)
23. Web Services Choreography Description Language (WS-CDL), Version 1.0 (November 2005), http://www.w3.org/TR/ws-cdl-10/
24. Web Services Flow Language (WSFL), http://xml.coverpages.org/wsfl.html
25. Web Service Choreography Interface (WSCI), Version 1.0 (January 2004), http://www.w3.org/TR/2002/NOTE-wsci-20020808
26. Massuthe, P., Reisig, W., Schmidt, K.: An operating guideline approach to the SOA. Annals of Mathematics, Computing & Teleinformatics 1(3), 35–43 (2005)
27. Zhou, L., Xiao, H., Ping, J., Pu, G., Zhang, H.: Simulation and Validation of Web Services Choreography. In: SOCA 2009, Taipei, China (December 2009)
28. Kaschner, K., Lohmann, N.: Automatic Test Case Generation for Interacting Services. In: ICSOC 2008. LNCS, vol. 5472, pp. 66–78. Springer, Heidelberg (2009)
29. Z3: An Efficient SMT Solver, http://research.microsoft.com/en-us/um/redmond/projects/z3/

Applying Ordinary Differential Equations to the Performance Analysis of Service Composition*

Zuohua Ding[1], Hui Shen[1], and Jing Liu[2]

[1] Center of Math Computing and Software Engineering
Zhejiang Sci-Tech University
Hangzhou, Zhejiang, 310018, China
[2] Software Engineering Institute
East China Normal University
Shanghai, 200062, China

Abstract. Web services technology has yet to address questions such as how can I know that the Web service will meet my performance requirements such as response time? In this paper, a new method is proposed to measure the performance of service composition. Service composition described with BPEL is modeled by a family of ordinary differential equations, where each equation describes the state change of the service composition. Each service state is measured by a time-dependent function that indicates the extent to which the state can be reached in execution. This measure information can help us to conduct performance analysis such as estimating response time, throughput and efficiency. This method has the following advantages: 1) it treats the system as a 'white' box and displays a global picture of execution state to the users, thus users know exactly where to improve the performance; 2) it can entirely avoid state explosion problem; 3) it is faster than SPN based performance analysis methods.

Keywords: Service composition, BPEL, performance analysis, Petri net, ordinary differential equation.

1 Introduction

In service-oriented computing (SOC), developers use services as fundamental elements in their application-development processes. To create applications, SOC developers use service composition. Developers can then solve complex problems by combining available basic services. Service composition thus accelerates rapid application development, service reuse, and complex service consummation. While service composition brings us advantages, it also raises a question:

* This work is supported by the NSF under Grant No. 90818013, No.90718014, 973 Program 2009CB320702, Shanghai TCSM No. 08510700300, and Zhejiang NSF under Grant No.Z1090357.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 155–170, 2010.
© Springer-Verlag Berlin Heidelberg 2010

how to evaluate the composite service? In other words, how to measure the system performance with quantitative characteristics such as system execution throughput?

Different approaches have been proposed in the literature for system performance analysis. Most of them exploit analytical models whose analysis is based on Petri net and Markov Theory. Stochastic Petri Nets (SPN) [2] are among the most popular modeling formalisms that have been used in the past decade. However, the state space explosion, which entails the solution of the underlying Markov Chain with a large number of states, limits the applicability of SPN based methods for analyzing large-scale systems even though many reduction skills have been proposed.

This paper presents a new method for performance analysis of service composition. The analytical model is also based on the (discrete) Petri net which has been used to model the service composition. Instead of developing reduction skill, we apply some relaxation skill developed by David and Alla[4] on the Petri net. This relaxation leads to a continuous-time formalism: Continuous Petri Net(CPN). The semantics of a continuous Petri net is defined by a set of Ordinary Differential Equations (ODEs). Hence, a service composition can be described by a family of ordinary differential equations, where each equation describes a state change of the service. A state can be measured by nonnegative number, called State Measure, indicating how much the state can be reached while the service is in execution. This information can help us to do performance analysis such as response time, throughput and efficiency.

Considering it is hard to find explicit analytic solutions for the nonlinear ordinary differential equations, we turn to find numerical solutions instead. Runge-Kutta method has been used to find the numerical solutions. The computational error is $O(h^5)$, where h is the step size. The Complexity analysis shows that the complexity for SPN is exponential while the complexity for CPN is linear. Thus our method can avoid state explosion problem which may happen in SPN based method.

There exist several proposals for service composition, such as BPEL, Semantic Web (OWL-S), Web Components, Algebraic Process Composition, Petri Nets[14]. In this paper, we choose BPEL to describe service composition. As the example, Diagnosis Apply of Regional Health Information System (RHIS) has been employed to illustrate our method.

This paper is organized as the following. Section 2 gives Petri net presentation for service composition. Section 3 models service composition with ordinary differential equations. In Section 4, by comparing with the SPN based analysis methods, we show how to perform analysis based on the solutions of ODEs. Section 5 is the complexity analysis for both CPN and SPN based analysis methods. Section 6 is the case study: Diagnosis Apply of Regional Health Information System (RHIS). Section 7 is a discussion of the related work. The last section, Section 8 is the conclusion of the paper.

2 Petri Net Representation of Service Composition

Definition 1. *A Petri net is a directed bipartite graph that can be written as a tuple* $N = (P, T, A, M_0)$*, where* P *is the set of places,* T *is the set of transitions,* $A \subset (P \times T) \cup (T \times P)$ *is the set of arcs, and* M_0 *is its initial marking.*

A Web service behavior is basically a partially ordered set of operations. Therefore, it is straightforward to map it into a Petri net. Operations are modeled by transitions and the state of the service is modeled by places. The arrows between places and transitions are used to specify causal relations. The initial marking indicates the start state.

BPEL is an XML language that supports service composition and can be used to describe executable business process behaviors. BPEL process is viewed as a series of activities which can be composed into complex activities by data flow and control flow. The rules rules for the Petri net presentation of control flows are similar to those in[10]. Next we slightly describe the rules for data flows. Here are some basic data flows for BPEL: $<\mathsf{invoke}>$, $<\mathsf{receive}>$, $<\mathsf{reply}>$.

The following is a format of *invoke*:

$$
\begin{array}{l}
<\mathsf{invoke} \\
\quad partnerLink = "insuranceA" \\
\quad portType = "ins : ComputeInsurancePremiumPT" \\
\quad operation = "ComputeInsurancePremium" \\
\quad inputVariable = "InsuranceRequest" / >
\end{array}
$$

It has one method *ComputeInsurancePremium* with only input variable *InsuranceRequest*. This method is asynchronous. *invoke* also has another synchronous format if it has input and output variables as the following:

$$
\begin{array}{l}
<\mathsf{invoke} \\
\quad partnerLink = "insuranceA" \\
\quad portType = "ins : ComputeInsurancePremiumPT" \\
\quad operation = "ComputeInsurancePremium" \\
\quad inputVariable = "InsuranceRequest" \\
\quad outputVariable = "InsuranceAResposne" / >
\end{array}
$$

Similarly, we can determine the communication types of other data flows. Thus in BPEL, the communication between services are through Synchronous or Asynchronous message passing. We design the Petri net representation to these data flow as shown in Fig. 1. In the figure, (a) is for asynchronous message passing mechanism, and (b) is for synchronous message passing mechanism, where A and B represent two services.

After translation of both control and data flows, we get a special class of Petri nets, called Message Passing Petri Net, or briefly MP. The following is the definition.

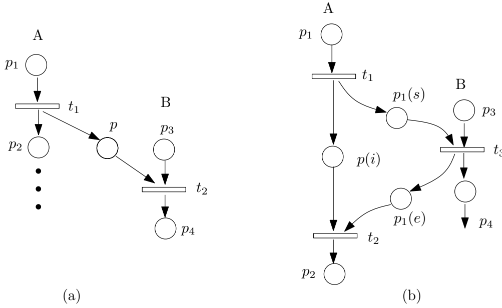


Fig. 1. Petri net for (a) asynchronous message passing, (b) synchronous message passing

Definition 2. *A Petri net P is MP if*

- *P has finitely many places;*
- *the places of P are partitioned into two disjoint partitions C and B;*
- *each place from C has one or two input transitions and one or two output transitions, but can not have two input transitions and two output transitions at the same time; and*
- *each place from B has one pair of input transitions and output transition*
- *each transition has one input place from C and one output place from C; and*
- *each transition has either*
  1) no input places from B and no output places from B; or
  2) no input places from B and one output place from B; or
  3) one input place from B and no output places from B; or
  4) one input place from B and one output place from B.
  Here C stands for "Internal States", while B stands for "Buffer".

In a MP, the Internal States and the directly connected transitions form one or several place/transition cycles, namely service net. Hence a MP can also be described as service nets that interact to each other through Buffer B, so is a BPEL process.

3 Modeling Service Composition with ODEs

3.1 Continuous Petri Net

Definition 3. *A* Continuous Petri Net *is a tuple* $CPN =< P, T, A, M_0, \lambda >$*, where* (P, T, A, M_0) *is a underlying untimed Petri net:* $P = \{p_1, p_2, \ldots, p_n\}$ *is the set of places,* $T = \{t_1, t_2, \ldots, t_m\}$ *is the set of transitions,* $A \subset (P \times T) \cup (T \times P)$ *is the set of arcs, and* M_0 *is the initial marking.* $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ *is a set of average firing rates of transitions.*

In the definition, the average firing rate λ_t is actually the reciprocal of firing delay of transition t, whcih can be obtained from requirements or from implementation. In a CPN, the marking of a place is no longer an integer but a nonnegative real number.

Definition 4. *Let $I = [0, \infty)$ be the time interval and let $m_i : I \to [0, \infty), i = 1, 2, \ldots, n$ be a set of mappings that associated with place p_i. A marking of a Continuous Petri Net $CPN =< P, T, A, M_0, \lambda >$ is a mapping*
$m : I \to [0, \infty)^n$,
$m(\tau) = (m_1(\tau), m_2(\tau), \ldots, m_n(\tau))$.

Definition 5. *(State Measure) Given any time moment $t \in [0, \infty)$, the marking value in a place is called the State Measure of this place, denoted as $m(t)$. State measures take nonnegative real numbers as their values.*

A transition is enabled if all the input places have nonzero markings. Only enabled transitions can be fired. So, if sme marking is moved into a place, we say that the state measure in this place is increasing; if some marking is moved out from a place, we say that the state measure in this place is decreasing. The change rate of state measure can be calculated as the following.

Let p_1 and p_2 be the input places of a transition t and their markings are $m_1(\tau)$ and $m_2(\tau)$, respectively. Let λ_t be the firing rate associated with t, then following the definition of Continuous Petri net defined by David and Alla [4], the marking moving out from p_1 and p_2 is defined by $\lambda_t \times min\{m_1(\tau), m_2(\tau)\}$. If t has only one input p_1, then marking $\lambda_t \times m_1(\tau)$ will be moved out from p_1.

3.2 Building Ordinary Differential Equation Model

Based on the semantics defined above, the state measure at each place can be calculated from an ordinary differential equation. We have the following cases.

1) One place to one place. As Fig.2 shows, place p will get marking from place p_1. Let the marking at place p and p_1 be m and m_1, respectively. Assume that the firing rates at transition t_1 and t are d_1 and d, respectively. Then the state measure m can be represented as

$$m'(\tau) = d_1 \times m_1(\tau) - d \times m(\tau). \quad (1)$$

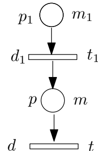


Fig. 2. One place to one place model

2) Two place to one place. As Fig. 3 shows, place p will get marking from place p_1 and p_2. Let the markings at place p_1, p_2 and p be m_1, m_2 and m, respectively. Assume that the firing rates at transition t_1 and t are d_1 and d, respectively. Then the state measure m can be represented as

$$m'(\tau) = d_1 \times min\{m_1(\tau), m_2(\tau)\} - d \times m(\tau). \quad (2)$$

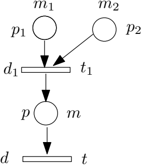


Fig. 3. Two places to one place model

3) One place to two places. As Fig. 4 shows, place p will get marking from place p_1, but will send some marking out together with place p_2. Let the markings at place p_1, p_2 and p be m_1, m_2 and m, respectively. Assume that the firing rates at transition t_1 and t are d_1 and d, respectively. Then the state measure m can be represented as

$$m'(\tau) = d_1 \times m_1(\tau) - d \times min\{m(\tau), m_2(\tau)\}. \tag{3}$$

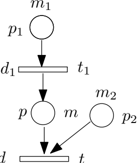


Fig. 4. One place to two places model

4) Two places to two places. As Fig. 5 shows, transition t_1 has two input places m_1 and m_2 and transition t has two input places m and m_3. Assume the firing rates at transition t_1 and t are d_1 and d, respectively. Then the marking m can be represented as

$$m'(\tau) = d_1 \times min\{m_1(\tau), m_2(\tau)\} - d \times min\{m(\tau), m_3(\tau)\}. \tag{4}$$

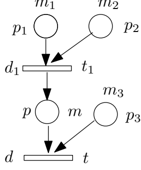


Fig. 5. Two places to two places model

From the differential equational model, we see that state measures are uniquely determined by the system structure and the firing rates. In general these equations can be classified as six types:

- Type 1 [Internal]. $m_i' = d_{i-1\times}m_{i-1} - d_i \times m_i$. Here m_i and m_{i-1} are the states of the same service net.
- Type 2 [Input-before]. $m_i' = d_{i-1} \times min\{m_{i-1}, m_k\} - d_i \times m_i$. Here m_i and m_{i-1} are the states of the same service net. m_k is the input to this service net from buffer.
- Type 3 [Input-after]. $m_i' = d_{i-1} \times m_{i-1} - d_i \times min\{m_i, m_k\}$. Here m_i and m_{i-1} are the states of the same service net. m_k is the input to this service net from buffer.
- Type 4 [Input-before-after]. $m_i' = d_{i-1} \times min\{m_{i-1}, m_k\} - d_i \times min\{m_i, m_l\}$. Here m_i and m_{i-1} are the states of the same service net. m_k and m_l are the inputs to this service net from buffer.
- Type 5 [Asynchronous]. $m_k' = d_i \times m_i - d_{i'} \times min\{m_{i'}, m_k\}$. Here m_i and $m_{i'}$ are the states of two different service nets respectively. m_k is the message between these two service nets.
- Type 6 [Synchronous]. $m_k' = d_i \times min\{m_i, m_l\} - d_{i'} \times min\{m_{i'}, m_k\}$. Here m_i and $m_{i'}$ are the states of two different service nets respectively. m_k and m_l are the messages between these two service nets, where m_l is usually indicates the request that can be calculated by Type 5 and m_k is the reply.

For the existence of the solutions of the above differential equations, we refer to[6].

4 Performance Analysis with ODEs

4.1 Performance Analysis with SPN

For the comparison, we briefly introduce the performance analysis method with stochastic Petri net. The following definition comes from [15].

Definition 6. *A* continuous stochastic Petri net *is a tuple* $SPN{=}(P, T, A, M_0, \lambda)$, *where* (P, T, A, M_0) *is a underlying untimed Petri net:* $P = \{p_1, p_2, \ldots, p_n\}$ *is the set of places,* $T = \{t_1, t_2, \ldots, t_m\}$ *is the set of transitions,* $A \subset (P \times T) \cup (T \times P)$ *is the set of arcs, and* M_0 *is the initial marking.* $\lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_m\}$ *is a set of average firing rates of transitions satisfying exponential distributions:*

$$\forall t \in T,\, F_t(x) = P\{X_t \leq x\} = 1 - e^{-\lambda_t x},$$

where x *represents time,* X_t *is a continuous random variable representing the time delay for transition* t, λ_t *is the average firing rate associated with transition* t.

A Markov process is a stochastic process that satisfies the *Markovian property*

$$P\{X(\tau) \leq x | X(t), t \in [0, \theta]\} = P\{X(\tau) \leq x | X(\theta) = y\},$$

for any $\tau > \theta$. Markov processes with a discrete state space are called Markov chains. If the parameter t is continuous, the process is a continuous-time Markov chain (CTMC). The time spent in states of a CTMC is a random variable with nonnegative exponential probability density function(pdf).

In practice, a CTMC is described through either a state transition rate diagram or a transition rate matrix, denoted by Q. The state transition rate diagram is a labelled directed graph whose vertices are labelled with the CTMC states, and whose arcs are labelled with the rate of the exponential distribution associated with the transition from a state to another.

If we have the reachable graph (or coverability tree) of SPN, then replacing the firing transition t associated with the arc by average firing rate λ_t (or marking related λ_t), we will get the CTMC that is isomorphic to SPN. Actually we have the following result[13].

Theorem 7. *Any SPN with finite places and finite transitions is isomorphic to a Continuous Time Markov Chain.*

This result enables us to compute average numbers of tokens in places of SPN, from which most of the performance indexes can be derived.

4.2 Numbers of Average Tokens of SPN = State Measures of CPN

Both SPN and CPN describe the events of the system, execution time of events, and the relations between events. In both models, the average execution time of events represent the firing rates of the corresponding transitions. The difference is that in SPN, the average number of tokens are obtained by computing Markov chain, while in CPN, the state measures are obtained by solving ordinary differential equations. However, we have the following result.

Theorem 8. *If SPN and CPN model the same system, then the average numbers of tokens in places of SPN equal the state measures in places of CPN.*

Proof. Our proof is on the basis of[11]. We consider three situations.

(1) Net with single input.

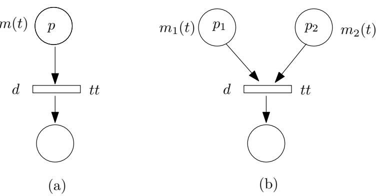


Fig. 6. A transition with single input

Fig. 6(a) shows a transition with a single input. In the figure, p is the input place of transition tt, $m(t)$ represents the state measure of p at time t, d is the average firing rate of tt, W is the time delay of tt. Assume that p has initial value k_1 at time t_0. In SPN, a transition needs time from enabled to firing, and this time is represented by a random variable W, which is subjected to an

exponential distribution function: $F_{tt}(\Delta t) = P[W \leq \Delta t] = 1 - e^{-d\Delta t}$, $\Delta t > 0$. Assume that after Δt time, the average number of tokens of p is k_1', then

$$k_1' = E[m(p)] = k_1 P[W > \Delta t] = k_1 e^{-d\Delta t}.$$

In CPN, for this net, we have equation $x' = -dx$ and $x(t_0) = k_1$. Solving this equation, we get $m(t_0 + \Delta t) = k_1 e^{-d\Delta t}$.

Thus in this case, our result is correct.

(2) Net with two inputs. Fig. 6(b) shows a transition with two input places. In the figure, p_1 and p_2 are two input places of transition tt, $m_1(t)$ and $m_2(t)$ represent state measures of p_1 and p_2 at time t, respectively. d is the average firing rate of tt, W is the delay of tt. Assume that p_1 has initial value k_1 at time t_0 and p_2 has initial value k_2 at time t_0.

In SPN, let k_1' and k_2' be the new average numbers of tokens of p_1 and p_2 after time Δt, respectively. Then

$$\begin{aligned} k_1' &= E[m(p_1)] \\ &= k_1 - min\{k_1, k_2\} P[W \leq \Delta t] \\ &= k_1 - min\{k_1, k_2\}(1 - P[W > \Delta t]) \\ &= k_1 - min\{k_1, k_2\}(1 - e^{-d\Delta t}). \end{aligned}$$

In CPN, for this net, we have equations:

$$x_1' = -d \times min\{x_1, x_2\},\ x_2' = -d \times min\{x_1, x_2\}$$

with the initial values: $x_1(t_0) = k_1, x_2(t_0) = k_2$. Since $min\{x_1, x_2\}$ is nondeterministic at every time t, it is hard to give analytic solution. With numerical solution, we may partition Δt into several small intervals Δt_i, such that on each such interval, $min\{x_1, x_2\}$ will take a fixed value. Now we solve the equations on interval Δt_1 as the follows.

i) If $x_1(t_0) < x_2(t_0)$, or $k_1 < k_2$, then $x_1' = -dmin\{x_1, x_2\} = -dx_1$, and $k_1' = x_1(t_0 + \Delta t_1) = k_1 e^{-d\Delta t_1}$.

ii) If $x_1(t_0) \geq x_2(t_0)$, or $k_1 \geq k_2$, then $x_1' = -dx_2$, $x_2' = -dx_2$, then $x_2(t_0 + \Delta t_1) = k_2 e^{-d\Delta t_1}$. Since in the steady state, the output markings from p_1 and p_2 are equal, and the output marking from p_2 is $k_2 - k_2 e^{-d\Delta t_1} = k_2(1 - e^{-d\Delta t_1})$, so the output marking from p_1 should be $k_2(1 - e^{-d\Delta t_1})$. Thus $k_1' = k_1 - k_2(1 - e^{-d\Delta t_1})$.

Combining i) and ii), we get $k_1' = k_1 - min\{k_1, k_2\}(1 - e^{-d\Delta t_1})$, which is the same as the average marking of SPN. We will get similar results on other intervals. Hence in this case, our result is also correct.

(3) Now consider the general cases for Fig. 2, Fig.3, Fig.4, and Fig.5. In SPN, for $\forall t \in T$, its marking flow rate, i.e. average marking moving to the output place in unit time, is $R(t, s) = W(t, s) \times \sum_{M \in E} P(M) \times \lambda$, where E is the set of all reachable markings that make t enable, λ is the average firing rate of t, $W(t, s)$ is the weight attached to the arc from transition t to place s. From the proof of (1) and (2), we know that $R(t, s) = d \times m$, or $R(t, s) = d \times min\{m_1, m_2\}$, where m, m_1, m_2 are the input places of tt, and d is the average firing rate.

Hence for the place m in Fig. 2, Fig.3, Fig.4, and Fig.5, at time t, *the average number of markings of a place = average number of markings to this place - average number of markings out this place*, i.e., $\int_0^t (d_1 m_1 - dm)dt$ for Fig.2; $\int_0^t (d_1 min\{m_1, m_2\} - dm)dt$ for Fig.3; $\int_0^t (d_1 m_1 - dmin\{m_2, m\})dt$ for Fig.4; $\int_0^t (d_1 min\{m_1, m_2\} - dmin\ \{m_3, m\})dt$ for Fig.5. These expressions are actually the solutions of Equation (1), Equation (2), Equation (3), and Equation (4) in Section 3.2, respectively. Hence, *the average number of tokens of m in SPN = state measure value of min CPN.*

Hence, we complete the proof.

Based on the above result, we can perform analysis of service composition using the solutions of ordinary differential equations.

4.3 Measures of Performance

We consider four performance indexes in this paper: response time, throughput, resource utilization rate and the maximum load of the system.

(1) Response Time. Response time is the time that cost in the process, which is from the beginning that the patients request to the end that the system responses to the patient. Thus the response time is actually a response between patients and a subsystem. We consider the problem to calculate response time as a queuing theory problem, and then we could calculate it using Little Rule and Flow Balance Principle. Thus the response time is the sum of all the average number of state measures in the subsystem (queue) divided by the average marking flow velocity of the subsystem.

(2) Throughput: Throughput is the number of patient requests that the subsystem can accept or deal with in a given time period.

(3) Resource Utilization Rate. Resource utilization rate is the extension of resource utilization, which can be explained based on the state measure of CPN. In the Petri net model, based on the state descriptions, we need to select a place as the measuring place. Using the state measure of this place, we can analyze the efficiency of the system.

(4) The maximum Load. System bottleneck problem is of interest to designers and users. Especially in real-time system, compared to the average throughput of the system, people pay more attention to the peak throughput. We can obtain this number by increasing the loads to the subsystem, and check if the marking stream into the subsystem is approaching to a steady value, then we can get the maximum number of the concurrent users of the system.

5 Complexity Analysis

Generally speaking, it is hard to find explicit analytic solutions for nonlinear ordinary differential equations, thus most of the time, we turn to find numerical solutions instead. We may use function *ode*45 in Matlab to solve our equations.

Function $ode45$ is the implementation of combined fourth and fifth-order Runge-Kutta method.

Numerical solutions may give us computational errors due to the algorithm and the machine. In our situation, the computation error can come from two sources: truncation error (because a truncated Taylor Series is used in the computation), and rounding error (because a finite number of binary digits is used inside the machine).

For the truncation error, the global truncation error of $ode45$ is $O(h^5)$ [8], where h is the step size,. Regarding the rounding error, since implicit Runge-Kutta method has stable area [8], and the algorithm is guaranteed to converge in the stable area. Thus the rounding error of the perturbation can not increase and will decrease to 0 in the iteration process [1].

For the complexity of Runge-Kutta method, if the accuracy is lower than 0.00001, then Runge-Kutta method is more efficient than Newton method. We know that the complexity for Newton method in general is $O(mn^3)$, where n is the number of variables and m is the iteration, which is usually $O(n)$ and never exceeds $O(n^2)$ [17]. Hence, the complexity for Runge-Kutta method in general is $O(n^4)$ and never exceeds $O(n^5)$.

Thus solving the state measure needs time $O(n^5)$, which is polynomial, where n is the number of equations. However, in SPN method, the the complexity to find the average number of tokens in a single place is $O(a^{p(k,n)}) + O(a^{p(k,n)}) = 2O(a^{p(k,n)})$, where k is the maximum value of the range of initial markings, and $p(k,n)$ is a polynomial of k and n. Hence to find average number of tokens in n places, the complexity is $n \times (O(a^{p(k,n)}) + O(a^{p(k,n)})) = nO(a^{p(k,n)})$.

6 Case Study: Diagnosis Apply of Regional Health Information System (RHIS)

We consider Diagnosis Apply of Regional Health Information System (RHIS) as the example. RHIS is an online platform through which the registered patients can reserve and apply diagnosis from doctors and experts. We build BPEL process model by IBM WebSphere Integration Developer (WID). WID can automatically generate BPEL file.

After translation, we get the MP representation as shown in Fig. 7. To illustrate our method, we have overlooked some details. The system has three service nets, Patient (Inputs to the system), Controller (Processing Diagnosis Apply), and Hospital (Patient Info Checking). The system works as following. The registered patients send the requests of remote medical care and electronic-health records (EHR) to the control system. After the controller receives the requests and EHR, it sends this information to the hospital to determine whether this patient is a legally registered user. Then the hospital will check the patient and send the result to the controller. Once the controller receives the result, it sends a message to the patient notifying him the status of the application.

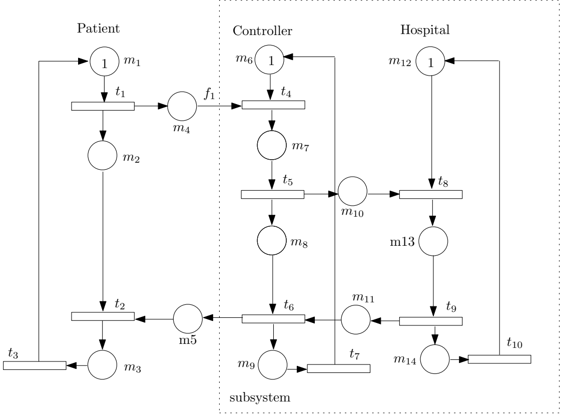


Fig. 7. Petri net representation of diagnosis apply

The corresponding differential equation model is:

$$(*)\begin{cases} m_1' = d_3 \times m_3 - d_1 \times m_1 \\ m_2' = d_1 \times m_1 - d_2 \times min(m_2, m_5) \\ m_3' = d_2 \times min(m_2, m_5) - d_3 \times m_3 \\ m_4' = d_1 \times m_1 - d_4 \times min(m_4, m_6) \\ m_5' = d_6 \times min(m_8, m_{11}) - d_2 \times min(m_2, m_5) \\ m_6' = d_7 \times m_9 - d_4 \times min(m_4, m_6) \\ m_7' = d_4 \times min(m_4, m_6) - d_5 \times m_7 \\ m_8' = d_5 \times m_7 - d_6 \times min(m_8, m_{11}) \\ m_9' = d_6 \times min(m_8, m_{11}) - d_7 \times m_9 \\ m_{10}' = d_5 \times m_7 - d_8 \times min(m_{10}, m_{12}) \\ m_{11}' = d_9 \times m_{13} - d_6 \times min(m_8, m_{11}) \\ m_{12}' = d_{10} \times m_{14} - d_8 \times min(m_{10}, m_{12}) \\ m_{13}' = d_8 \times min(m_{10}, m_{12}) - d_9 \times m_{13} \\ m_{14}' = d_9 \times m_{13} - d_{10} \times m_{14} \end{cases}$$

with the initial values: $m_1(0) = m_6(0) = m_{12}(0) = 1$, all other are 0.

With some simulation data, we get $d_1 = 0.017, d_2 = 0.008, d_3 = 0.017, d_4 = 0.033, d_5 = 0.028, d_6 = 0.014, d_7 = 0.05, d_8 = 0.022, d_9 = 0.033$, and $d_{10} = 0.05$. To illuminate the meanings of d_i, take d_1 for example. Since $d_1 = 0.017$, it means that about $0.017 \times 60 = 1.02 \approx 1$ users sending request out per minute in the average state. We also assume that the average number of requests from patients is 1/minute.

With the help of Matlab, we get all the solutions of m_i as shown in Fig. 8.

When $t > 1000s$, all the solutions are approaching to steady values, i.e. $m_1 \approx 0.13, m_2 \approx 0.74, m_3 \approx 0.13, m_4 \approx 0.07, m_5 \approx 0.27, m_6 \approx 0.56, m_7 \approx 0.08, m_8 \approx 0.32, m_9 \approx 0.04, m_{10} \approx 0.10, m_{11} \approx 0.16, m_{12} \approx 0.89, m_{13} \approx 0.07, m_{14} \approx 0.04$.

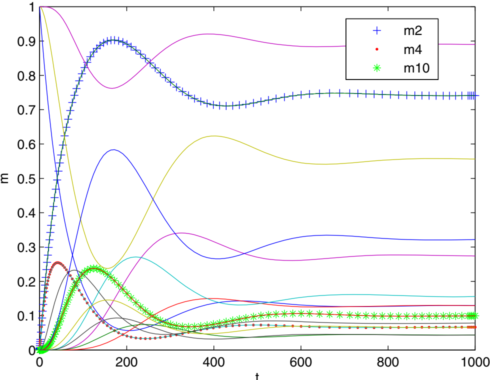


Fig. 8. State measures of diagnosis apply

Based on the calculation result, we give the performance analysis in the following.

(1) Response time. We regard the controller and the hospital as the subsystems. The response time is the queue length of the task divided by the average marking velocity of the subsystem, which equals to the number of marking entering the subsystem per unit time in steady state. The queue length of the task is the sum of the average number of marking in each place in the subsystem. Thus the response time $T = (m_6 + m_7 + \cdots + m_{14})/(d_1 m_1)$. From Fig.9, we find that when $t > 1000s$, the response time is approaching to a steady value: $T = 2.2555/(1 \times 0.1296) \approx 17minutes$.

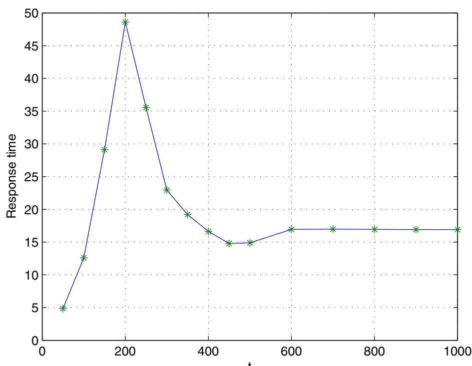


Fig. 9. The response time of diagnosis apply

(2) Throughput. From Fig.10, we see that when $t > 1000s$, the throughput of the subsystem is approaching to a steady value: $0.1296 \times 1 \times 60 \approx 8$, it means that the subsystem can accept 8 users per hour.

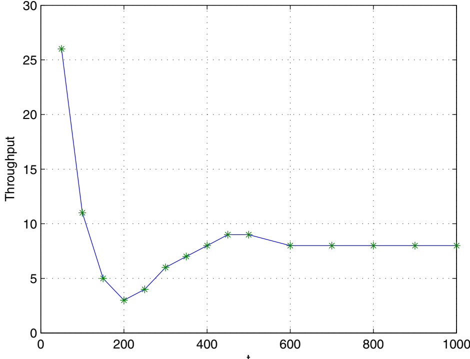


Fig. 10. The throughput of diagnosis apply

(3) Resource utilization rate. State m_2 stands for 'patient waiting for response'. We may choose m_2 as the measuring place. When $t > 1000s$, the value of m_2 is approaching to stable state, i.e. $m_2 \approx 0.74$, we imply that most of the patients are waiting for the response. Thus, the system efficiency at this moment is $(1 - 0.74) = 0.26 = 26\%$, which is bad. From Fig. 8, we find the reason is that very little patient requests ($m_4 \approx 0.07$) have been sent out to the controller, and very little patient information($m_{10} \approx 0.10$) has been sent to the hospital for checking.

(4) The maximum load. Because we have assumed that the average number of requests from patients is 1/minute, $N = 1$ means that 1 user logins to the system. Increasing loads N, we analyze the peak throughput of the system. It can be seen from Fig.11 that when $N > 3$, the number of marking entering the subsystem will approach to a steady value no matter how the loads are increased. So the system can serve $3 \times 1 = 3$ patients per minute to send requests at the same time.

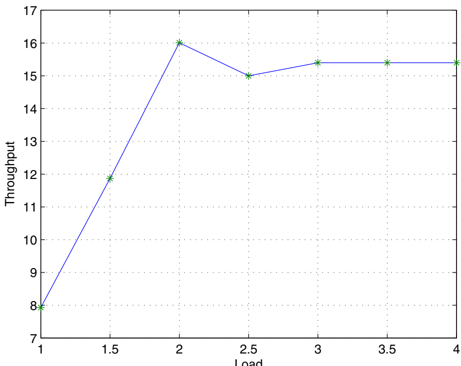


Fig. 11. The load of diagnosis apply

7 Related Work

Many efforts have been done in the performance analysis. Even through different analysis models have been proposed, eventually Markov theory will get involved. So the state explosion problem is still there. Using our method, this problem can be solved. Moreover, the analysis speed is faster. Here are some examples.

The most related work is from Tan et al. [16]. They used Stochastic Petri Net to model web services described by WSDL. Based on numeric compression and decomposition techniques, they developed a set of reduction techniques, which focus on five basic structures in web service flow: sequential, parallel, conditional, loop and mutex. As their experiment shows, the state space is reduced, however, the state space explosion problem is still existed.

Dong et al. [7] proposed an analytical approach to predict the performance of web service composition built on BPEL. The approach first translates web service composition specification into Stochastic Petri Nets, then from the SPN model and the corresponding continuous-time Markov chain, they derived the analytical performance estimates of process-completion-time. Datla and Popstojanova [5] measured the performance at architectural level. They studied the web services performance by recording the component execution events in the application server logs and developing scripts in AWK scripting language to automate the task of extracting response times for each component from the application server logs. The workload is described with a Discrete Time Markov Chain (DTMC) which characterizes the customers request patterns. Lin et al.[12] gave performance analysis for workflow management system. The system is modeled by stochastic Petri net, namely workflow model (WF-SPN), which is the extension of WF-net. Chandrasekaran et al. [3] described Service Composition and Execution Tool (SCET) and various methodologies that could be adopted for evaluating the performance of a Web process.

8 Conclusion

Service composition described by BPEL can be described by a set of ordinary differential equations, and its performance such as response time, throughput and system efficiency can be analyzed based on the solutions of the set of equations. Our experiments show that we may get better performance by adjusting the firing rates. In the future work, we will give the general rules to adjust the firing rates to get desired performance. We need to mention that although our method is developed for service composition, this method can also be used for the performance analysis of other time dependent systems.

References

1. Ascher, U.M., Petzold, L.R.: Computer methods for ordinary differential equations and differential-algebraic equations. Society for Industrial & Applied Mathematis, Philadelphia, PA, USA (1998)
2. Bause F., Kritzinger F.: Stochastic Petri Nets-An Introduction to the Theory. Vieweg Verlag (2002)

3. Chandrasekaran, S., Miller, J.A., Silver, G.S., Arpinar, B., Sheth, A.P.: Performance analysis and simulation of composite web services. Electronic Markets 13(2), 120–132 (2003)
4. David R., Alla H.: Continuous Petri nets. In: Proceedings of the 8th European Workshop on Application and Theory of Petri nets, pp. 275–294 (1987)
5. Datla, V., Popstojanova, K.G.: Measurement-based performance analysis of e-commerce applications with web services components. In: Proceedings of IEEE International Conference on e-Business Engineering, pp. 305–314 (2005)
6. Ding, Z.: Static analysis of concurrent programs using ordinary differential equations (Invited Speech). In: Leucker, M., Morgan, C. (eds.) Theoretical Aspects of Computing - ICTAC 2009. LNCS, vol. 5684, pp. 1–35. Springer, Heidelberg (2009)
7. Dong, Y., Xia, Y., Zhu, Q., Huang, Y.: A Stochastic Approach to Predict Performance of Web Service Composition. In: Proceedings of The 2nd International Symposium on Electronic Commerce and Security, pp. 460–464 (2009)
8. Hairer, E., Nørsett, S.P., Wanner, G.: Solving Ordinary Differential Equations(I)(II). In: Nonstiff Problems, 2nd edn. Springer, Heidelberg (1993)
9. Harrow, A., Hassidim, A., Lloyd, S.: Quantum algorithm for solving linear systems of equations. Phys. Rev. Lett. 103(15), 150502 (2009)
10. Hinz, S., Schmidt, K., Stahl, C.: Transforming BPEL to Petri nets. In: van der Aalst, W.M.P., Benatallah, B., Casati, F., Curbera, F. (eds.) BPM 2005. LNCS, vol. 3649, pp. 220–235. Springer, Heidelberg (2005)
11. Hiraishi, K.: Performance evaluation of workflows using continuous Petri nets with interval firing speeds. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 231–250. Springer, Heidelberg (2008)
12. Lin, C., Qu, Y., Ren, F., Marinescu, D.C.: Performance equivalent analysis of workflow systems based on stochastic Petri net models. In: Han, Y., Tai, S., Wikarski, D. (eds.) EDCIS 2002. LNCS, vol. 2480, pp. 1–64. Springer, Heidelberg (2002)
13. Molloy, M.K.: On the integration of delay and throughput measures in distributed processing models. Ph.D. dissertation, University of California, Los Angeles (1981)
14. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Computing 8, 51–59 (2004)
15. Molloy, M.K.: Performance analysis using stochastic Petri nets. IEEE Transactions on Computers C-31(9), 913–917 (1982)
16. Tan, Z., Lin, C., Yin, H., Hong, Y., Zhu, G.: Approximate performance analysis of web services flow using stochastic Petri net. In: Jin, H., Pan, Y., Xiao, N., Sun, J. (eds.) GCC 2004. LNCS, vol. 3251, pp. 193–200. Springer, Heidelberg (2004)
17. Teukolsky, S.A., Press, W.H., Vetterling, W.T.: Numerical Recipes in C++, 2nd edn. Cambridge Univ. Press, Cambridge (1993)

Verifying Heap-Manipulating Programs with Unknown Procedure Calls

Shengchao Qin[1], Chenguang Luo[2,⋆], Guanhua He[2],
Florin Craciun[1], and Wei-Ngan Chin[3]

[1] Teesside University, Middlesbrough TS1 3BA, UK
[2] Durham University
[3] National University of Singapore

Abstract. Verification of programs with invocations to unknown procedures is a practical problem, because in many scenarios not all codes of programs to be verified are available. Those unknown calls also pose a challenge for their verification. This paper addresses this problem with an attempt to verify the full functional correctness of such programs using pointer-based data structures. Provided with a Hoare-style specification $\{\Phi_{pr}\}$ prog $\{\Phi_{po}\}$ where program prog contains calls to some unknown procedure unknown, we infer a specification $mspec_u$ for unknown from the calling contexts, such that the problem of verifying prog can be safely reduced to the problem of proving that the procedure unknown (once its code is available) meets the derived specification $mspec_u$. The expected specification $mspec_u$ for the unknown procedure unknown is automatically calculated using an abduction-based shape analysis specifically designed for a combined abstract domain. We have also done some experiments to validate the viability of our approach.

1 Introduction

While automated verification of heap-manipulating programs remains a big challenge [16], significant advances have been seen recently since the emergence of separation logic [13]. For instance, SpaceInvader [3] can verify the pointer safety of a large portion of the Linux kernel and many device drivers using shared mutable data structures; THOR [10] employs additional numerical analysis to help gain better precision for data structure properties such as list length; Hip/Sleek [11] can verify more sophisticated properties involving both shape and numerical information, such as sortedness, height-balanced and red-black properties. These are all successful examples of verification/analysis of heap-manipulating programs, esp. those processing pointer-based shared mutable data structures.

However, a recent prevalent trend of component-based software engineering [7] poses great challenge for quality assurance and verification of programs. This methodology involves the integration of software components from both

⋆ Now with Citigroup Inc.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 171–187, 2010.
© Springer-Verlag Berlin Heidelberg 2010

native development and third-parties, and thus the source code of some components/procedures might be unknown for verification. For example, some programs may have calls to third-party library procedures whose code is not accessible (e.g. in binary form). Some components may be invoked by remote procedure calls only with a native interface such as COM/DCOM [14]. Still, some components could be used for dynamic upgrading of running systems whose cost of being stopped/restarted is too expensive to bear [15]. Other scenarios include function pointers (e.g. in C), interface method invocation (e.g. in OO) and mobile code, which all contain procedures not available for static verification.

To verify such programs, existing approaches generally do not provide elegant solutions. For example, black-box testing [2] regards the unknown procedures as black-boxes to test their functionality, which cannot formally prove the absence of program bugs, therefore may not be enough for safety-critical systems. Likewise, specification mining [1] discovers possible specifications for the (unknown part of the) program by observing its execution and traces, which is also dynamically performed and bears the same problem. For static verifiers/analysers, SpaceInvader [3] simply assumes the program and the unknown procedure have disjoint memory footprints so that the unknown call can be safely ignored due to the hypothetical frame rule [12], whereas this assumption does not hold in many cases. Some methods [4,6] try to take into account all possible implementations for the unknown procedure; however there can be too many such candidates in general, and hence the verification might be infeasible for large-scaled programs. Finally, some verifiers will just stop at the first unknown procedure call and provide an incomplete verification [11], which is obviously undesirable.

Approach and contributions. We propose a novel approach in this paper to verifying heap-manipulating programs calling unknown procedures. Given a specification $\mathsf{S} = \{\Phi_{pr}\}$ prog $\{\Phi_{po}\}$ where prog contains calls to an unknown procedure unknown, we try to infer a specification S_u for unknown based on the calling context(s) of prog. The verification of prog against S can now be safely reduced to the verification of unknown against the inferred specification S_u, provided that the verification of the known fragments does not cause any problems. The inferred specification is subject to a later verification when an implementation or a specification for the unknown procedure becomes available. This is essentially an improvement of our previous work [8] by extending the program properties to be verified from simple pointer safety to full functional correctness of linked data structures. Such properties include structural numerical ones like size and height, relational numerical ones like sortedness, and multi-set ones like symbolic content. Our paper makes the following technical contributions:

- We propose a novel framework in a combined abstract domain (involving both shape and pure properties) for the verification of full functional correctness of programs with unknown calls.
- Our approach is essentially *top-down*, as it can be used to infer the specification for callee procedures based on the specification for the caller procedure. Hence it may benefit the general software development process as a complement for current *bottom-up* approaches [3,11].

– We have invented an abduction mechanism which can be applied in this combined domain. It not only can infer shape-based anti-frames for an entailment, but also can discover corresponding pure information (numerical and/or multi-set) as well. We also defined a partial order as a guidance for the quality of abduction results.
– We have conducted some initial experimental studies to test the viability and performance of our approach. Preliminary results show that our approach can derive expressive specifications which fully capture the behaviours of the unknown code in many cases.

In the following we will first illustrate our approach with an illustrative example and then describe its formal settings. Any technical details not described due to space limit can be found in our technical report [9].

2 The Approach

We first introduce our specification mechanism, followed by an illustrative example for the verification.

2.1 User-Defined Predicates

Separation logic [13] extends Hoare logic to support reasoning about shared mutable data structures. It provides separation conjunction ($*$) to form formulae like $\mathtt{p_1} * \mathtt{p_2}$ to assert that two heaps described by $\mathtt{p_1}$ and $\mathtt{p_2}$ are domain-disjoint. Our abstract domain is founded on a hybrid logic of both separation logic and classical first-order logic to specify both separation and pure properties. Over this domain we allow user-defined inductive predicates. For example, with a data structure definition for a node in a list `data node { int val; node next; }`, we can define a predicate for a list with the content stored in its nodes as

$$\mathtt{root}::\mathtt{llB}\langle\mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee (\exists \mathtt{v},\mathtt{q},\mathtt{S_1} \cdot \mathtt{root}::\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle * \mathtt{q}::\mathtt{llB}\langle\mathtt{S_1}\rangle \wedge \mathtt{S}{=}\mathtt{S_1} \sqcup \{\mathtt{v}\})$$

The parameter `root` for the predicate `llB` is the root pointer referring to the list. Its content is denoted by the multi-set `S`. A uniform notation $\mathtt{p}::\mathtt{c}\langle\mathtt{v}^*\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node, the notation represents a singleton heap, $\mathtt{p} \mapsto \mathtt{c}[\mathtt{v}^*]$, e.g. the $\mathtt{root}::\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\mathtt{v}^*$, e.g., the $\mathtt{q}::\mathtt{llB}\langle\mathtt{S_1}\rangle$ above.

If users want to verify a sorting algorithm, they can incorporate sortedness property into the above predicate as follows:

$$\begin{array}{l}\mathtt{sllB}\langle\mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee \\ \quad (\mathtt{root}::\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle * \mathtt{q}::\mathtt{sllB}\langle\mathtt{S_1}\rangle \wedge \mathtt{S}{=}\{\mathtt{v}\} \sqcup \mathtt{S_1} \wedge (\forall \mathtt{u} {\in} \mathtt{S_1} \cdot \mathtt{v} {\leq} \mathtt{u}))\end{array}$$

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and $\mathtt{S_1}$, are implicitly existentially quantified. Meanwhile, later we may still use underscore _ to denote an implicitly quantified variable. Such user-supplied predicates can be used to specify method specifications.

2.2 Illustrative Example

In this section, we illustrate informally, via an example, how our approach verifies a program by inferring the specification for the unknown procedure it invokes.

Example 1 (Motivating example). Our goal is to verify the procedure `sort` against the given specification shown in Figure 1. According to the specification, the procedure takes in a non-empty linked list `x` and returns a sorted list referenced as `res`. The (symbolic) content of these two lists are identical (`S`). Note that `sort` calls an unknown procedure `unknown` at line `4`. As we do not have available knowledge about it, the discovery of its specifications is essential for both the verification and our understanding of the program (such that we may find out what sorting algorithm this procedure implements).

We conduct a forward analysis on the program body starting with the precondition $\texttt{x::llB}\langle\texttt{S}\rangle$ (line `0`). The results of our analysis (e.g. the abstract states) are marked as comments in the code. The analysis carries on until it reaches the unknown procedure call at line `4`.

As afore-shown, the current state before line `4` is $\texttt{x::llB}\langle\texttt{S}\rangle\wedge\texttt{x}\neq\texttt{null}$ (σ at line `3a`). Then we want to discover the precondition for the unknown call from it. To do that, we split σ into two disjoint parts: the local part $\Phi^{\texttt{u}}_{pr}$ (line `3c`) that is depended on, and possibly mutated by, the unknown procedure; and the frame part $\texttt{R}_0$ (line `3e`) that is not accessed by the unknown procedure. Intuitively, the local part of a state w.r.t. a set of variables `X` is the part of the heap reachable from variables in `X`; while the frame part denotes the unreachable heap part. Thus we take $\Phi^{\texttt{u}}_{pr}$ (line `3c`) as a crude precondition for the unknown procedure. The frame part $\texttt{R}_0$ is not touched by the unknown call and will remain in the post-state, as shown in line `4c`.

At line `4c`, the abstract state after the unknown call (σ) consists of two parts: one is the aforesaid frame $\texttt{R}_0$ not accessed by the call, and the other is the procedure's postcondition which is unfortunately not available. Our next step is to discover the postcondition by examining the code fragment after the unknown call (lines `4a` to `8e`). For this task, a traditional approach is a backward reasoning from the caller's postcondition towards the unknown call's postcondition. However, this is proven infeasible for separation logic based shape domain by previous works [3], and hence we employ another approach with a forward reasoning from the unknown call towards the caller's postcondition, using *abduction* to discover the unknown call's postcondition.

Initially, we assume the unknown procedure having an empty heap σ'_0 as its postcondition[1], and gradually discover the missing parts of the postcondition during the symbolic execution of the code fragment after the unknown call. To do that, our analysis keeps track of a pair (σ, σ') at each program point, where σ refers to the current heap state, and σ' denotes the expected postcondition discovered so far for the unknown procedure. The notations $\sigma'_{\texttt{i}}$ are used to represent parts of the discovered postcondition.

[1] Note that we introduce fresh logical variables `a` and $\texttt{res}_{\texttt{u}}$ to record the value of `x` and `y` when `unknown` returns.

```
0 node sort(node x) requires x::llB⟨S⟩ ensures res::sllB⟨S⟩
1 {  // res is the value returned by the procedure
1a   // Forward analysis begins with current state σ : x::llB⟨S⟩
2   if (x == null) return null;
2a   // σ : x::llB⟨S⟩ ∧ x=null ∧ res=null
2b   // Check whether current state meets the postcondition: σ ⊢ res::sllB⟨S⟩
2b   //  which succeeds; the verification on this branch terminates
3   else {
3a   // σ : x::llB⟨S⟩ ∧ x≠null
3b   // Unknown call is now encountered (line 4); extract its precondition from σ:
3c   // Φ^u_pr := Local(σ, {x}) := x::llB⟨S⟩ ∧ x≠null
3d   // Also distinguish the frame part not touched by unknown call:
3e   // R_0 := Frame(σ, {x}) := emp ∧ x≠null
4   node y = unknown(x);
4a   // Immediately after the unknown call we know nothing about its effect, so
4b   // we begin to discover its post-effect starting from emp (saved in σ'):
4c   // σ'_0 : emp ∧ x=a ∧ y=res_u        σ := R_0 ∗ σ'_0        σ' := σ'_0
4d   // Next instruction (y.next) requires y be a node
4e   // But the entailment checking σ ⊢ y::node⟨v, p⟩ fails
4f   // This requirement might be part of the unknown call's post-effect; we use
4g   // abduction to find it and add it to current state and unknown call's post:
4h   // σ ∗ [σ'_1] ▷ y::node⟨v, p⟩ (s.t. σ ∗ σ'_1 ⊢ y::node⟨v, p⟩ ∗ true)
4i   // σ'_1 : y::node⟨v, p⟩                 σ := σ ∗ σ'_1        σ' := σ' ∗ σ'_1
5   node z = y.next;
5a   // Current state σ : y::node⟨v, z⟩
5b   // Next instruction invokes this procedure recursively and requires its pre, but
5c   //  σ ⊢ z::llB⟨S_1⟩ fails possibly due to lack of knowledge about unknown call
5d   // Again we use abduction to find the missing part of unknown call's post-effect
5e   // σ ∗ [σ'_2] ▷ z::llB⟨S_1⟩ (s.t. σ ∗ σ'_2 ⊢ z::llB⟨S_1⟩ ∗ true)
5f   // σ'_2 : z::llB⟨S_1⟩                   σ := σ ∗ σ'_2        σ' := σ' ∗ σ'_2
6   node w = sort(z);
6a   // Current state σ : y::node⟨v, z⟩ ∗ w::sllB⟨S_1⟩ (w already refers to a sorted list)
7   y.next = w;
7a   // Current state σ : y::node⟨v, w⟩ ∗ w::sllB⟨S_1⟩
8   return y;
8a   // σ : y::node⟨v, w⟩ ∗ w::sllB⟨S_1⟩ ∧ res=y; it should imply sort's postcondition
8b   // But σ ⊢ res::sllB⟨S⟩ still fails, suggesting more post-effect of unknown call
8c   // A final abduction is conducted to find it: σ ∗ [σ'_3] ▷ res::sllB⟨S⟩
8d   // σ'_3 : S={v}⊔S_1 ∧ ∀u∈S_1·v≤u   σ := σ ∗ σ'_3        σ' := σ' ∗ σ'_3
8e   // All abduction results will be combined at last to form unknown call's post
9   } }
9a // Φ^u_pr : a::llB⟨S⟩ ∧ a≠null (a is the unknown procedure's formal parameter)
9b // Φ^u_po : res_u::node⟨v, b⟩ ∗ b::llB⟨S_1⟩ ∧ S={v}⊔S_1 ∧ ∀u∈S_1·v≤u
```

Fig. 1. Verification of `sort` which invokes an unknown procedure `unknown`

At line 5, y.next is dereferenced, whose value is then assigned to z. Such dereference causes a problem, as we have an empty heap beforehand (σ in line 4c). However, this is not necessarily due to a program error; it might be attributed to the fact that the unknown call's postcondition is still unknown. Therefore, our analysis performs an abduction (line 4h) to infer the missing part σ'_1 for σ such that $\sigma * \sigma'_1$ implies that y points to a node. As shown in line 4i, σ'_1 is inferred to be $\mathtt{y}{::}\mathtt{node}\langle\mathtt{v},\mathtt{p}\rangle$, which is accumulated into σ' as part of the expected postcondition of the unknown procedure. (We will explain the details for abduction in Section 4.) Now the heap state combined with the inferred σ'_1 meets the requirement of the dereference, and thus the forward analysis continues.

At line 6, the procedure sort is called recursively. Here the current heap state still does not satisfy the precondition of sort (as shown in line 5c). Blaming the lack of knowledge about the unknown call's postcondition, we conduct another abduction (line 5e) to infer the missing part σ'_2 for σ such that $\sigma * \sigma'_2$ entails the precondition of sort w.r.t. some substitution $[\mathtt{z}/\mathtt{x}]$. Updated with the abduction result $\mathtt{z}{::}\mathtt{llB}\langle\mathtt{S_1}\rangle$, the program state now meets the precondition of sort, which is later transformed to $\mathtt{w}{::}\mathtt{sllB}\langle\mathtt{S_1}\rangle$ as the effect of sorting over z.

After that, line 7 links y and the sorted list w together. Then y is returned as the procedure's result at last. The corresponding state σ at line 8a is expected to establish the postcondition of sort for the overall verification to succeed. However, it does not (as shown in line 8b). Again this might be because part of the unknown call's postcondition is still missing. Therefore, we perform a final abduction (line 8c) to infer the missing σ'_3 as follows:

$$(\mathtt{y}{::}\mathtt{node}\langle\mathtt{v},\mathtt{w}\rangle * \mathtt{w}{::}\mathtt{sllB}\langle\mathtt{S_1}\rangle \wedge \mathtt{res}{=}\mathtt{y}) * [\sigma'_3] \triangleright \mathtt{res}{::}\mathtt{sllB}\langle\mathtt{S}\rangle$$

such that $\sigma * \sigma'_3$ implies the postcondition. In this case, our abductor returns σ'_3, a sophisticated pure constraint $\mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S_1} \wedge \forall \mathtt{u}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\leq}\mathtt{u}$, as the result which is then added into σ', as shown in line 8d.

Finally, we generate the expected pre/post-specification for the unknown procedure (lines 9a and 9b). The precondition is obtained from the local pre-state of the unknown call, $\Phi^{\mathtt{u}}_{pr}$ at line 3c, by replacing all variables that are aliases of a with the formal parameter a. The postcondition is obtained from the accumulated abduction result, σ', after performing a similar substitution (which also involves formal parameter $\mathtt{res_u}$). Our discovered specification for the unknown procedure node unknown(node a) is:

$$\Phi^{\mathtt{u}}_{pr} : \mathtt{a}{::}\mathtt{llB}\langle\mathtt{S}\rangle \wedge \mathtt{a}{\neq}\mathtt{null}$$
$$\Phi^{\mathtt{u}}_{po} : \exists \mathtt{b} \cdot \mathtt{res_u}{::}\mathtt{node}\langle\mathtt{v},\mathtt{b}\rangle * \mathtt{b}{::}\mathtt{llB}\langle\mathtt{S_1}\rangle \wedge \mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S_1} \wedge \forall \mathtt{u}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\leq}\mathtt{u}$$

This derived specification has two implications. The first is that the entire program is verified on the condition that unknown meets such specification. The second is an improvement of our understanding on the behaviours of both the caller (sort) and the callee (unknown): the callee should choose the smallest element from its input list, and its way of choice decides the type of sorting for the caller (selection or bubble sort).

$$
\begin{array}{llll}
\mathit{Prog} & ::= \mathit{tdecl}\ \mathit{meth}\ \mathit{munk} & \mathit{tdecl} ::= \mathit{datat} \mid \mathit{spred} & \\
\mathit{datat} & ::= \texttt{data}\ c\ \{\ \mathit{field}\ \} & \mathit{field} ::= t\ x & t ::= c \mid \tau \\
\mathit{meth} & ::= t\ mn\ ((\boldsymbol{t}\ \boldsymbol{x});(\boldsymbol{t}\ \boldsymbol{y}))\ \mathit{mspec}\ \{e\} & \tau ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} & \\
\mathit{munk} & ::= t\ mn\ ((\boldsymbol{t}\ \boldsymbol{x});(\boldsymbol{t}\ \boldsymbol{y}))\ \mathit{mspec}\ \{v\} & & \\
e & \multicolumn{3}{l}{::= d \mid d[x] \mid x{=}e \mid e_1;e_2 \mid t\ x;\ e \mid \texttt{if}\ (x)\ e_1\ \texttt{else}\ e_2 \mid \texttt{while}\ x\ \{e\}\ \texttt{inv}\ \Delta} \\
u & \multicolumn{3}{l}{::= unk(\boldsymbol{x};\boldsymbol{y}) \mid unk(\boldsymbol{x_0};\boldsymbol{y_0}); e_1; unk_1(\boldsymbol{x_1};\boldsymbol{y_1}); e_2; ...; e_{n-1}; unk_n(\boldsymbol{x_n};\boldsymbol{y_n}) \mid} \\
 & \multicolumn{3}{l}{\texttt{if}\ (x)\ v\ \texttt{else}\ e \mid \texttt{if}\ (x)\ e\ \texttt{else}\ v \mid \texttt{if}\ (x)\ v_1\ \texttt{else}\ v_2 \mid \texttt{while}\ x\ \{v\}\ \texttt{inv}\ \Delta} \\
v & \multicolumn{3}{l}{::= e_1; u; e_2} \\
d & \multicolumn{3}{l}{::= \texttt{null} \mid k^{\tau} \mid x \mid \texttt{skip} \mid \texttt{new}\ c(\boldsymbol{x}) \mid mn(\boldsymbol{x};\boldsymbol{y})} \\
d[x] & \multicolumn{3}{l}{::= x.f \mid x.f{:=}z \mid \texttt{free}(x)}
\end{array}
$$

Fig. 2. A core (C-like) imperative language

3 Language and Abstract Domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Figure 2. A program *Prog* consists of two parts: type declarations and method declarations. The type declarations *tdecl* can define either data type *datat* (e.g. `node`) or predicate *spred* (e.g. `llB`). The method declarations include *meth* and *munk*, of which the second contains invocations to unknown procedures while the first does not. The *spred* and *mspec* are defined in Figure 3.

Note that the language is expression-oriented, so the body of a method is an expression composed of standard instructions and constructors of an imperative language. e is the (recursively defined) program constructor and d and $d[x]$ are atom instructions. Note also that the language allows both call-by-value and call-by-reference method parameters (which are separated with a semicolon ; where the ones before ; are call-by-value and the ones after are call-by-reference).

To address the unknown calls, we employ *unknown constructors* u and v to denote expressions that involve invocations to the unknown procedures ($unk(\boldsymbol{x}, \boldsymbol{y})$). An *unknown block* v is defined as a sequence of normal expressions sandwiching an *unknown expression* u, which can be a single unknown call, or a sequence of unknown calls, or an if-conditional statement/while loop containing an unknown block. Our aim is to discover the specifications for the unknown procedures in u and v to verify the whole program.

$$
\begin{array}{lll}
\mathit{mspec} & ::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} & \mathit{spred} ::= \texttt{root}{::}c\langle \boldsymbol{v}\rangle \equiv \Phi \\
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta & \\
\Phi & ::= \bigvee \boldsymbol{\sigma} & \sigma ::= \exists \boldsymbol{v} \cdot \kappa \wedge \pi \\
\kappa & ::= \texttt{emp} \mid v{::}c\langle \boldsymbol{v}\rangle \mid \kappa_1 * \kappa_2 & \pi ::= \gamma \wedge \phi \\
\gamma & \multicolumn{2}{l}{::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \texttt{true} \mid \gamma_1 \wedge \gamma_2} \\
\phi & \multicolumn{2}{l}{::= \varphi \mid b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi} \\
b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 & a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & \multicolumn{2}{l}{::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) \mid |\texttt{B}|} \\
\varphi & \multicolumn{2}{l}{::= v{\in}\texttt{B} \mid \texttt{B}_1{=}\texttt{B}_2 \mid \texttt{B}_1{\sqsubset}\texttt{B}_2 \mid \texttt{B}_1{\sqsubseteq}\texttt{B}_2 \mid \forall v{\in}\texttt{B}{\cdot}\phi \mid \exists v{\in}\texttt{B}{\cdot}\phi} \\
\texttt{B} & \multicolumn{2}{l}{::= \texttt{B}_1{\sqcup}\texttt{B}_2 \mid \texttt{B}_1{\sqcap}\texttt{B}_2 \mid \texttt{B}_1{-}\texttt{B}_2 \mid \emptyset \mid \{v\}}
\end{array}
$$

Fig. 3. The specification language

Our specification language (in Figure 3) allows (user-defined) shape predicates to specify both separation and pure properties. The shape predicates *spred* are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [11].

A conjunctive abstract program state σ is composed of a heap (shape) part κ and a pure part π, where π consists of γ, ϕ and φ as aliasing, numerical and bag information, respectively. We use SH to denote a set of such conjunctive states. During our verification, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ (and the set of such disjunctions $\mathcal{P}_{\mathsf{SH}}$). An abstract state Δ can be normalised to the Φ form [11].

The memory model of our specification formulae is adapted from the model given for "early versions" of separation logic [13], except that we have extensions to handle user-defined shape predicates and related pure properties. Meanwhile, for program variables in abstract states, we use unprimed ones to denote their initial values and primed ones for current values [9,11].

4 Abduction

As shown in Section 2, when analysing the code after an unknown call, it is possible that the current state cannot meet the required precondition for the next instruction due to the lack of information about the unknown procedure. Therefore we need to infer the unknown procedure's specification with *abduction* (or abductive reasoning) [3,5]. It works as follows: for a failed entailment checking $\sigma_1 \vdash \sigma_2 * \mathtt{true}$, it attempts to compute an anti-frame σ', such that $\sigma_1 * \sigma' \vdash \sigma_2 * \mathtt{true}$ succeeds. For instance, the entailment checking $\mathtt{emp} \vdash \mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle$ fails as the antecedent contains an empty heap. Then $\mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle$ will be found to strengthen the antecedent and validate the entailment $\mathtt{emp} * \mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle \vdash \mathtt{x}{::}\mathtt{llB}\langle\mathtt{S}\rangle$.

An abduction $\sigma_1 * [\sigma'] \triangleright \sigma_2$ can also be written as $\sigma_1 * [\sigma'] \triangleright \sigma_2 * \sigma_3$, where σ_1 and σ_2 are inputs, σ' is the abduction result (the anti-frame), and σ_3 is the frame part resulted from the entailment checking $\sigma_1 * \sigma' \vdash \sigma_2$.

Our abduction rules given in Figure 4 deal with four different cases. The first rule triggers when the LHS (σ) does not imply the RHS (σ_1) but the RHS implies

$$\frac{\sigma \nvdash \sigma_1 * \mathtt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \triangleright \sigma_1 * \sigma_2}$$

$$\frac{\begin{array}{c}\sigma \nvdash \sigma_1 * \mathtt{true} \quad \sigma_1 \nvdash \sigma * \mathtt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data_no}(\sigma_0) \leq \mathsf{data_no}(\sigma_1) \\ \sigma_0 \vdash \sigma_1 * \sigma' \text{ or } \sigma_0 * [\sigma_0'] \triangleright \sigma_1 * \sigma' \quad \sigma''{=}\mathit{XPure}_1(\sigma') \quad \sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2\end{array}}{\sigma \wedge [\sigma''] \triangleright \sigma_1 * \sigma_2}$$

$$\frac{\sigma \nvdash \sigma_1 * \mathtt{true} \quad \sigma_1 \nvdash \sigma * \mathtt{true} \quad \sigma_1 * [\sigma_1'] \triangleright \sigma * \sigma' \quad \sigma''{=}\mathit{XPure}_1(\sigma') \quad \sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2}{\sigma \wedge [\sigma''] \triangleright \sigma_1 * \sigma_2}$$

$$\frac{\sigma \nvdash \sigma_1 * \mathtt{true} \quad \sigma_1 \nvdash \sigma * \mathtt{true} \quad \sigma * \sigma_1 \nvdash \mathtt{false}}{\sigma * [\sigma_1] \triangleright \sigma_1 * \sigma_2}$$

Fig. 4. Abduction rules

the LHS with some formula (σ') as the frame. This rule is quite general and applies in many cases, such as the state immediately after an unknown call where we start with $\texttt{emp}$ as the heap state. For the example above $\texttt{emp} \nvdash \texttt{x::llB}\langle\texttt{S}\rangle$, the RHS can entail the LHS with frame $\texttt{x::llB}\langle\texttt{S}\rangle$. The abduction then checks whether σ plus the frame information σ' entails σ_1 with some frame formula σ_2 ($\texttt{emp}$ in this example), and returns the result $\texttt{x::llB}\langle\texttt{S}\rangle$.

In the case described by the second rule, neither side implies the other, e.g. for $\texttt{x::sllB}\langle\texttt{S}\rangle$ as LHS (σ) and $\exists \texttt{p}, \texttt{u}, \texttt{v} \cdot \texttt{x::node}\langle\texttt{u}, \texttt{p}\rangle * \texttt{p::node}\langle\texttt{v}, \texttt{null}\rangle$ as RHS (σ_1). As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like $\texttt{sllB}$), its certain disjunctive branches may imply σ_1. As the rule suggests, to accomplish abduction $\sigma * [\sigma''] \rhd \sigma_1 * \sigma_2$, we first unfold σ ($\sigma_0 \in \mathsf{unroll}(\sigma)$) and try entailment or further abduction with the results (σ_0) against σ_1. If it succeeds with a frame σ', then we first obtain a pure approximation of σ' with *XPure* [11], and confirm the abduction by ensuring $\sigma \wedge \sigma'' \vdash \sigma_1 * \sigma_2$, for some σ_2. For the example above, the abduction returns $|\texttt{S}|{=}2$ as the anti-frame σ' and discovers the nontrivial frame $\texttt{S}{=}\{\texttt{u}, \texttt{v}\} \wedge \texttt{u}{\leq}\texttt{v}$ (σ_2). Note the function $\mathsf{data_no}$ returns the number of data nodes in a state, e.g. it returns one for $\texttt{x::node}\langle\texttt{v}, \texttt{p}\rangle * \texttt{p::llB}\langle\texttt{T}\rangle$. This syntactic check is important for the termination of the abduction. The unroll unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{n} \sigma_i$), and returns the result as a set of formulae ($\{\sigma_1, ..., \sigma_n\}$). The *XPure* is a strengthened version of that in [11], as it also keeps the pure part of σ' in the result.

In the third rule, neither side entails the other, and the second rule does not apply, for example $\exists \texttt{p}, \texttt{u}, \texttt{v} \cdot \texttt{x::node}\langle\texttt{u}, \texttt{p}\rangle * \texttt{p::node}\langle\texttt{v}, \texttt{null}\rangle$ as LHS (σ) and $\exists \texttt{S} \cdot \texttt{x::sllB}\langle\texttt{S}\rangle$ as RHS (σ_1). In this case the antecedent cannot be unfolded as they are already data nodes. As the rule suggests, it reverses two sides of the entailment and applies the second rule to uncover the constraints σ_1' and σ'. Then it checks that the LHS (σ), with σ' added, does imply the RHS (σ_1) before it returns σ'. For the example above, the abduction returns $\texttt{u}{\leq}\texttt{v}$ which is essential for the two nodes to form a sorted list (σ_1).

When an abduction is conducted, the first three rules should be attempted first; if they do not succeed in finding a solution, the last rule is invoked to simply add the consequent to the antecedent, provided that they are consistent. It is effective for situations like $\texttt{x::node}\langle_, _\rangle \nvdash \texttt{y::node}\langle_, _\rangle$, where we should add $\texttt{y::node}\langle_, _\rangle$ to the LHS directly (as the other three rules do not apply here).

One observation on abduction is that there can be many solutions of the anti-frame σ' for the entailment $\sigma_1 * \sigma' \vdash \sigma_2 * \texttt{true}$ to succeed. For instance, $\texttt{false}$ is always a solution but should be avoided where possible. For all possible solutions to an abduction, we can compare their "quality" with a partial order $\preceq$ over SH defined by the entailment relationship ($\vdash$):

$$\sigma_1 \preceq \sigma_2 =_{df} \sigma_2 \vdash \sigma_1 * \texttt{true}$$

and the smaller (weaker) one in two abduction solutions is regarded as better. We prefer to find solutions that are (potentially locally) minimal with respect to $\preceq$ and consistent. However, such solutions are generally not easy to compute and could incur excess cost (with additional disjunction in the analysis). Therefore,

our abductive inference is designed more from a practical perspective to discover anti-frames that should be suitable as specifications for unknown procedures, and the partial order $\preceq$ is more a guidance of the decision choices of our abduction implementation, rather than a guarantee to find the theoretically best solution.

5 Verification

This section presents our algorithms to verify programs with unknown calls.

1. Main verification algorithm. Our main verification algorithm is given in Figure 5. It verifies an unknown block v (the third parameter) against given specifications $mspec_v$ (the second parameter). The first parameter includes the specifications of already available procedures which might be invoked as well as the unknown ones in the program to be verified. Upon successful verification, this algorithm returns specifications that should be met by the unknown procedures in v. If the verification fails, it suggests that the current program cannot meet one or more given specifications due to a potential program bug. The specifications for unknown procedures will be expressed in terms of special variables $\boldsymbol{a}, \boldsymbol{b}$, etc. as in the earlier example.

The algorithm initialises in the first two lines. It distinguishes the body of the unknown block v (as an unknown expression u in between two normal expressions e_1 and e_2), sets up the set to store discovered specifications (line 1), and finds the program variables that are potentially accessed by v and u, respectively ($\mathsf{prog_var}$ in line 2). Note that $\boldsymbol{x_0}$ and $\boldsymbol{x}$ are the variables read by v and u, and $\boldsymbol{y_0}$ and $\boldsymbol{y}$ are those mutated. For example, if v contains an assignment $\texttt{y} = \texttt{x}$ then $\texttt{x}$ will be in $\boldsymbol{x_0}$ and $\texttt{y}$ in $\boldsymbol{y_0}$.

After the initialisation, for each specification (*requires* Φ_{pr} *ensures* Φ_{po}) to verify against (line 3), the algorithm works in three steps. The first step is to compute the preconditions of u (lines 4–7). It first conducts a symbolic execution from Φ_{pr} over e_1 (the program segment before u) to obtain its post-states, from which the preconditions for u will be extracted (line 4). The symbolic execution is essentially a forward analysis whose details are presented later. If the post-states include $\texttt{false}$, then it means the given Φ_{pr} cannot guarantee e_1's memory safety, and thus fail is returned (line 5). Otherwise, each post-state of e_1 is processed by function Local as a candidate precondition for u (line 7). Intuitively, it extracts the part of each σ reachable from the variables that may be accessed by u, namely, $\boldsymbol{x}$ and $\boldsymbol{y}$. The function Local is defined as follows:

$$\begin{aligned}\mathsf{Local}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\}) =_{df}\ & \exists \mathtt{fv}(\kappa{\wedge}\pi){\cup}\{\boldsymbol{z}\} \setminus \mathtt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \cdot \\ & \mathtt{ReachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi\end{aligned}$$

where $\mathsf{fv}(\sigma)$ stands for all free (program and logical) variables occurring in σ, and $\mathtt{ReachVar}(\kappa \wedge \pi, \{\boldsymbol{x}\})$ is the minimal set of variables reachable from $\{\boldsymbol{x}\}$:

$$\begin{aligned}&\{\boldsymbol{x}\} \cup \{z_2 \mid \exists z_1, \pi_1 \cdot z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi, \{\boldsymbol{x}\}) \wedge \pi{=}(z_1{=}z_2 \wedge \pi_1)\} \cup \{z_2 \mid \\ &\quad \exists z_1, \kappa_1 \cdot z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi, v) \wedge \kappa{=}(z_1{::}c\langle .., z_2, ..\rangle * \kappa_1)\} \subseteq \mathtt{ReachVar}(\kappa{\wedge}\pi, \{\boldsymbol{x}\})\end{aligned}$$

Algorithm $\mathsf{Verify}(\mathcal{T}, mspec_v, v)$
1 Denote v as $\{e_1; u; e_2\}$; $mspec_u := \emptyset$
2 $(\boldsymbol{x_0}, \boldsymbol{y_0}) := \mathsf{prog_var}(v)$; $(\boldsymbol{x}, \boldsymbol{y}) := \mathsf{prog_var}(u)$
3 **foreach** $(requires\ \Phi_{pr}\ ensures\ \Phi_{po}) \in mspec_v$ **do**
4 $\quad \mathsf{S}_0 := [\![e_1]\!]_{\mathcal{T}}\{\Phi_{pr} \wedge \boldsymbol{y_0'}{=}\boldsymbol{y_0}\}$
5 $\quad$ **if** `false` $\in \mathsf{S}_0$ **then return** fail **endif**
6 $\quad$ **foreach** $\sigma \in \mathsf{S}_0$ **do**
7 $\quad\quad \Phi^u_{pr} := \mathsf{Local}(\sigma, \{\boldsymbol{x}, \boldsymbol{y}\})$
8 $\quad\quad \boldsymbol{z} := \mathsf{fv}(\Phi^u_{pr}) \setminus \{\boldsymbol{x}, \boldsymbol{y}\}$
9 $\quad\quad \mathsf{S} := [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{([\boldsymbol{b}/\boldsymbol{y}]\,\mathsf{Frame}(\sigma, \{\boldsymbol{x}, \boldsymbol{y}\}) \wedge \boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}, \mathsf{emp} \wedge \boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c})\}$
10 $\quad\quad \mathsf{S}' := \{\ (\sigma, \sigma') \mid (\sigma, \sigma'){\in}\mathsf{S} \wedge \sigma \vdash \Phi_{po} * \mathtt{true}\ \} \ \cup \{\ (\sigma * \sigma'', \sigma' * \sigma'') \mid (\sigma, \sigma'){\in}\mathsf{S} \wedge \sigma \nvdash \Phi_{po}{*}\mathtt{true} \wedge \sigma{*}[\sigma''] \rhd \Phi_{po}{*}\mathtt{true}\ \}$
11 $\quad\quad$ **if** $\exists(\sigma, \sigma'){\in}\mathsf{S}'\ .\ \mathsf{fv}(\sigma') \nsubseteq \mathsf{ReachVar}(\sigma, \{\boldsymbol{a}, \boldsymbol{b}\})$ **then return** (fail, σ') **endif**
12 $\quad\quad$ **foreach** $(\sigma, \sigma') \in \mathsf{S}'$ **do**
13 $\quad\quad\quad \Phi^u_{pr} := [\boldsymbol{a}/\boldsymbol{x}, \boldsymbol{b}/\boldsymbol{y}, \boldsymbol{c}/\boldsymbol{z}]\,\Phi^u_{pr}$
14 $\quad\quad\quad \Phi^u_{po} := \mathsf{sub_alias}(\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$
15 $\quad\quad\quad g := (\mathsf{fv}(\Phi^u_{pr}) \cap \mathsf{fv}(\Phi^u_{po})) \cup \{\boldsymbol{a}, \boldsymbol{b}\}$
16 $\quad\quad\quad mspec_u := mspec_u \cup \{(requires\ \exists(\mathsf{fv}(\Phi^u_{pr}) \backslash g) \cdot \Phi^u_{pr}\ ensures\ \Phi^u_{po})\}$
17 $\quad\quad$ **end foreach**
18 $\quad$ **end foreach**
19 **end foreach**
20 $\mathcal{T}_u := \mathsf{CaseAnalysis}(\mathcal{T}, mspec_u, u)$
21 **return** $\mathcal{T} \uplus \mathcal{T}_u$
end Algorithm

Fig. 5. The main verification algorithm

That is, it is composed of aliases of $\boldsymbol{x}$ as well as variables reachable from $\boldsymbol{x}$. And the formula $\mathtt{ReachHeap}(\kappa{\wedge}\pi, \{\boldsymbol{x}\})$ denotes the part of κ reachable from $\{\boldsymbol{x}\}$ and is formally defined as the $*$-conjunction of the following set of formulae:

$$\{\kappa_1 \mid \exists z_1, z_2, \kappa_2 \cdot z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi, \{\boldsymbol{x}\}) \wedge \kappa{=}\kappa_1{*}\kappa_2 \wedge \kappa_1{=}z_1{::}c\langle .., z_2, ..\rangle\}$$

The second step is to discover the postconditions for u (lines 9–11). This is mainly completed with another symbolic execution with abduction over e_2 (line 9), whose details are also introduced later. Here we denote u's post-state as `emp`, since its knowledge is not available yet. Therefore, the initial state for the symbolic execution of e_2 is simply the frame part of state not touched by u. The function Frame is formally defined as

$$\mathsf{Frame}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\}) =_{df} \exists \boldsymbol{z} \cdot \mathtt{UnreachHeap}(\kappa \wedge \pi, \{\boldsymbol{x}\}) \wedge \pi$$

where $\mathtt{UnreachHeap}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\})$ is the formula consisting of all $*$-conjuncts from κ which are not in $\mathtt{ReachHeap}(\exists \boldsymbol{z} \cdot \kappa \wedge \pi, \{\boldsymbol{x}\})$.

The conjunctions $\boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}$ in line 9 are to keep track of variable snapshot accessed by u using the special variables $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$. Then the symbolic

execution returns a set S of pairs (σ, σ') where σ is a possible post-state of e_2 and σ' records the discovered effect of u. However, maybe u still has some effect that is only exposed in the expected postcondition Φ_{po} for the whole program; therefore we need to check whether or not σ can establish Φ_{po}. If not, another abduction $\sigma*[\sigma''] \triangleright \Phi_{po}$ is invoked to discover further effect σ'' which is then added into σ'.

There can still be some complication here. Note that the effect discovered during e_2's symbolic execution may not be attributed all over to u; it is also possible that there is a bug in the program, or the given specification is not sufficient. As a consequence of that, the result σ' returned by our abduction may contain more information than what can be expected from u, in which case we cannot simply regard the whole σ' as the postcondition of u. To detect such a situation, we introduce the check in line 11. It tests whether the whole abduction result is reachable from variables accessed by u. If not, then the unreachable part cannot be expected from u, which indicates a possible bug in the program or some inconsistency between the program and its specification. In such cases, the algorithm returns an additional formula that can be used by a further analysis to either identify the bug or strengthen the specification.

The third step (lines 12–17) is to form the derived specifications for u in terms of variables $\boldsymbol{a}, \boldsymbol{b}$ and g. Here g denotes logical variables not explicitly accessed by u, but occurring in both pre- and postconditions (ghost variables). The formula $\mathsf{sub_alias}(\sigma', \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$ is obtained from σ' by replacing all variables with their aliases in $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$. Finally, at line 20, the obtained specifications $mspec_u$ for u are passed to the case analysis algorithm (given in Figure 6) to derive the specifications of unknown procedures invoked in u.

2. Case analysis algorithm. In order to discover specifications for unknown procedures invoked in u, the algorithm in Figure 6 conducts a case analysis according to the structure of u. In the first case (line 2), u is simply a single unknown call. In this situation, the algorithm returns all the pre-/postcondition pairs from $mspec_u$ as the unknown procedure's specifications.

In the second case (line 4), u is an `if`-conditional and both branches contain an unknown block. The algorithm uses the main algorithm to verify the two branches separately with preconditions $\Phi_{pr} \wedge x$ and $\Phi_{pr} \wedge \neg x$ respectively, where Φ_{pr} is one of the preconditions of the whole `if`. The results obtained from the two branches are then combined using the $\uplus$ operator:

$$R_1 \uplus R_2 =_{df} \{(\mathsf{f}, \mathsf{Refine}(mspec^1_{\mathsf{f}} \cup mspec^2_{\mathsf{f}})) \mid (\mathsf{f}, mspec^1_{\mathsf{f}}) \in R_1 \wedge (\mathsf{f}, mspec^2_{\mathsf{f}}) \in R_2\}$$

where Refine is used to eliminate any specification ($requires\ \Phi_{pr}\ ensures\ \Phi_{po}$) from a set if there exists a "stronger" one ($requires\ \Phi'_{pr}\ ensures\ \Phi'_{po}$) such that $\Phi'_{pr} \preceq \Phi_{pr}$ and $\Phi_{po} \preceq \Phi'_{po}$. It is defined as

$$\begin{array}{l} \mathsf{Refine}(\emptyset) =_{df} \emptyset \\ \mathsf{Refine}(\{(requires\ \Phi_{pr}\ ensures\ \Phi_{po})\} \cup \mathsf{Spec}) =_{df} \\ \quad \textbf{if}\ \exists (requires\ \Phi'_{pr}\ ensures\ \Phi'_{po}) \in \mathsf{Spec} \cdot \Phi'_{pr} \preceq \Phi_{pr} \wedge \Phi_{po} \preceq \Phi'_{po} \\ \quad \textbf{then}\ \mathsf{Refine}(\mathsf{Spec})\ \textbf{else}\ \{(requires\ \Phi_{pr}\ ensures\ \Phi_{po})\} \cup \mathsf{Refine}(\mathsf{Spec}) \end{array}$$

and $\uplus$ is to refine the union of two specification sets.

```
Algorithm CaseAnalysis(T, mspec_u, u)
 1 switch u
 2   case unk(x; y)
 3     return { (unk(x; y), mspec_u)}
 4   case if (x) v1 else v2
 5     mspec_T := {(requires Φ_pr∧x ensures Φ_po) |
                    (requires Φ_pr ensures Φ_po) ∈ mspec_u}
 6     mspec_F := {(requires Φ_pr∧¬x ensures Φ_po) |
                    (requires Φ_pr ensures Φ_po) ∈ mspec_u}
 7     R1 := Verify(T, mspec_T, v1)
 8     R2 := Verify(T, mspec_F, v2)
 9     return R1 ⊎ R2
10   case if (x) v else e
11     mspec_T := {(requires Φ_pr∧x ensures Φ_po) |
                    (requires Φ_pr ensures Φ_po) ∈ mspec_u}
12     R := Verify(T, mspec_T, v)
13     if ∃(requires Φ_pr ensures Φ_po) ∈ mspec_u, σ ∈ [[e]]_T{Φ_pr ∧ ¬x} ·
           σ=false ∨ σ ⊬ Φ_po∗true then return fail
14     else return R endif
15   case if (x) e else v   (Similar to the previous case)
16   case while x {v} inv Δ
17     return Verify(T, requires Δ∧x ensures Δ, v)
18   case unk_0(x_0; y_0) { ; e_i; unk_i(x_i; y_i)}_{i=1}^n
19     return { (unk_i(x_i; y_i), SeqUnkCalls(T, mspec_u, u))}_{i=0}^n
end Algorithm
```

Fig. 6. The case analysis algorithm

The third and fourth cases (lines 10 and 15) are for `if`-conditionals which contain only one unknown block in one of the two branches. This is handled in a similar way as in the second case. The only difference is, for the branch without unknown blocks, we need to verify it with the underlying semantics (line 13).

The fifth case is the `while` loop. As we assume its invariant is already given for the verification, we simply verify its body with the main algorithm, regarding the invariant as both pre- and postconditions (line 17).

In the last case (line 21), where u consists of multiple unknown procedure calls in sequence, another algorithm SeqUnkCalls is invoked to deal with it. We informally introduce its idea here due to space limit; its algorithm and subsequent discussions about our solution can be found in the report [9].

Suppose we have $\{\Phi_{pr}\}$ $\{unk_0(\boldsymbol{x_0}; \boldsymbol{y_0}); e; unk_1(\boldsymbol{x_1}; \boldsymbol{y_1})\}$ $\{\Phi_{po}\}$ to be verified, where e is the only known code fragment within the block. Our current solution finds a common specification to capture both unknown procedures' behaviours.

The algorithm works in three steps. In the first step, it extracts the precondition for the first procedure, say Φ^u_{pr}, from the given precondition Φ_{pr} by extracting the part of heap that may be accessed by the call via $\boldsymbol{x_0}$ and $\boldsymbol{y_0}$, which

is similar to the first step of the main algorithm Verify. Aiming at a general specification for both unknown calls, it then assumes that the second procedure has a similar precondition Φ^u_{pr}. In the second step, it symbolically executes the code fragment e with the help of the abductor, to discover a crude postcondition, say Φ^u, expected from the first unknown call. This is similar to the second step of the main algorithm Verify, except that the postcondition for e is now assumed to be Φ^u_{pr}. In the third step, the algorithm takes Φ^u (with appropriate variable substitutions) as the postcondition of the second unknown call, and checks whether or not the derived post (Φ^u) satisfies Φ_{po}. If not, it invokes another abduction to strengthen Φ^u to obtain the final postcondition Φ^u_{po} for the unknown procedures. Note that this strengthening does not affect soundness: the strengthened Φ^u_{po} can still be used as a general postcondition for both unknown procedures.

3. Abstract semantics. Our verification algorithms utilise two semantics: an underlying semantics and an abstract semantics with abduction. They are used to conduct the forward analysis over program body. The type of our underlying semantics is defined as

$$[\![e]\!] \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where AllSpec contains procedure specifications (extracted from the program *Prog*). For some expression e, given its precondition, the semantics will calculate the postcondition.

The abstract semantics with abduction is of the type:

$$[\![e]\!]^{\mathsf{A}} \ : \ \mathsf{AllSpec} \rightarrow \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \rightarrow \mathcal{P}(\mathsf{SH} \times \mathsf{SH})$$

It takes a piece of program and a specification table, to map a (disjunctive) set of pair of symbolic heaps to another such set (where the first in the pair is the current state and the second is the accumulated postcondition for unknown call).

Formal definition of both semantics can be found in the technical report [9].

4. Soundness and termination. For soundness of our verification, we have the following theorem:

Theorem 1 (Soundness). *Our analysis is sound due to the soundness of entailment checking, abduction and abstract semantics.*

The proof for entailment checking is by structural induction [11]. For abduction, as its result is always checked with entailment, its soundness follows that of entailment checking's. Finally, the soundness of abstract semantics is proven by induction over program constructors.

We have also confirmed that our verification terminates:

Theorem 2 (Termination). *Our verification will terminate in finite steps for finite input of programs and specifications.*

This is because our algorithms perform structural reasoning over finite input. More details of soundness and termination can be found in our report [9].

6 Experimental Results

We have implemented the verification algorithms and the abstract semantics with Objective Caml and evaluated them over some heap-manipulating programs. The results are in Tables 1 and 2. In each table, the first and second columns denote the programs used for evaluation and their time consumption, respectively. During the experiments, we manually hide some instructions in the original programs as calls to unknown procedures, whose specifications we try to discover during the verification process. Accordingly, the third column in the first table contain both the specifications of the programs to be verified (upper line), and the derived specifications for the unknown procedure (lower line). For the second table, as we used the same specification $\mathtt{x}::\mathtt{llB}\langle\mathtt{S}\rangle \ast\!\!\rightarrow \mathtt{res}::\mathtt{sllB}\langle\mathtt{S}\rangle$ to verify all the sorting algorithms, the third column (from the second line on) states the discovered specification for the unknown call only. Due to space limit, more experimental results are available in our report [9].

Table 1. Selected experimental results (lists and trees)

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$) and **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\rightarrow \Phi^u_{po}$)
List processing programs		
create	0.405	$\mathtt{emp} \wedge \mathtt{n}\geq 0 \ast\!\!\rightarrow \mathtt{res}::\mathtt{llB}\langle\mathtt{S}\rangle \wedge \mathtt{n}=\lvert\mathtt{S}\rvert \wedge \forall \mathtt{v}\in\mathtt{S}\cdot 1\leq\mathtt{v}\leq\mathtt{n}$
		$\mathtt{emp} \wedge \mathtt{a}\geq 1 \ast\!\!\rightarrow \mathtt{res}::\mathtt{node}\langle\mathtt{c},\mathtt{b}\rangle \wedge 1\leq\mathtt{c}\leq\mathtt{n}$
	1.020	$\mathtt{emp} \wedge \mathtt{n}\geq 0 \ast\!\!\rightarrow \mathtt{res}::\mathtt{sllB2}\langle\mathtt{S}\rangle \wedge \mathtt{n}=\lvert\mathtt{S}\rvert \wedge \forall \mathtt{v}\in\mathtt{S}\cdot 1\leq\mathtt{v}\leq\mathtt{n}$
		$\mathtt{emp} \wedge \mathtt{a}\geq 1 \ast\!\!\rightarrow \mathtt{res}::\mathtt{node}\langle\mathtt{c},\mathtt{b}\rangle \wedge \mathtt{a}-1\leq\mathtt{c}\leq\mathtt{a}$
sort_ insert	0.667	$\mathtt{x}::\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}\geq 1 \ast\!\!\rightarrow \mathtt{x}::\mathtt{ll}\langle\mathtt{m}\rangle \wedge \mathtt{m}=\mathtt{n}+1$
		$\mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{c}\rangle \ast \mathtt{c}::\mathtt{ll}\langle\mathtt{d}\rangle \ast\!\!\rightarrow \mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{e}\rangle \ast \mathtt{e}::\mathtt{ll}\langle\mathtt{d}+1\rangle$
	0.764	$\mathtt{x}::\mathtt{sll}\langle\mathtt{n},\mathtt{xs},\mathtt{xl}\rangle \wedge \mathtt{v}\geq\mathtt{xs} \ast\!\!\rightarrow \mathtt{x}::\mathtt{sll}\langle\mathtt{n}+1,\mathtt{mn},\mathtt{mx}\rangle \wedge \mathtt{mn}=\mathtt{xs} \wedge \mathtt{mx}=\mathtt{max}(\mathtt{xl},\mathtt{v})$
		$\mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{c}\rangle\ast\mathtt{c}::\mathtt{sll}\langle\mathtt{d},\mathtt{g},\mathtt{h}\rangle\wedge\mathtt{b}\leq\mathtt{f}\leq\mathtt{g} \ast\!\!\rightarrow \mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{e}\rangle\ast\mathtt{e}::\mathtt{sll}\langle\mathtt{d}+1,\mathtt{f},\mathtt{h}\rangle$
delete	0.646	$\mathtt{x}::\mathtt{llB}\langle\mathtt{S}\rangle \wedge \lvert\mathtt{S}\rvert\geq 2 \ast\!\!\rightarrow \mathtt{x}::\mathtt{llB}\langle\mathtt{T}\rangle \wedge \exists\mathtt{a}\cdot\mathtt{S}=\mathtt{T}\sqcup\{\mathtt{a}\}$
		$\mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{c}\rangle \ast \mathtt{c}::\mathtt{node}\langle\mathtt{d},\mathtt{e}\rangle \ast \mathtt{e}::\mathtt{llB}\langle\mathtt{E}\rangle \ast\!\!\rightarrow \mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{e}\rangle \ast \mathtt{e}::\mathtt{llB}\langle\mathtt{E}\rangle$
	0.916	$\mathtt{x}::\mathtt{sllB}\langle\mathtt{S}\rangle \wedge \lvert\mathtt{S}\rvert\geq 2 \ast\!\!\rightarrow \mathtt{x}::\mathtt{sllB}\langle\mathtt{T}\rangle \wedge \exists\mathtt{a}\cdot\mathtt{S}=\mathtt{T}\sqcup\{\mathtt{a}\}$
		$\mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{c}\rangle \ast \mathtt{c}::\mathtt{node}\langle\mathtt{d},\mathtt{e}\rangle \ast \mathtt{e}::\mathtt{sllB}\langle\mathtt{E}\rangle \wedge \forall\mathtt{f}\in\mathtt{E}\cdot\mathtt{b}\leq\mathtt{d}\leq\mathtt{f} \ast\!\!\rightarrow$ $\mathtt{a}::\mathtt{node}\langle\mathtt{b},\mathtt{e}\rangle \ast \mathtt{e}::\mathtt{sllB}\langle\mathtt{E}\rangle \wedge \forall\mathtt{f}\in\mathtt{E}\cdot\mathtt{b}\leq\mathtt{f}$
travrs	0.272	$\mathtt{x}::\mathtt{ll}\langle\mathtt{m}\rangle \wedge \mathtt{n}\geq 0\wedge\mathtt{m}\geq\mathtt{n} \ast\!\!\rightarrow \mathtt{x}::\mathtt{ls}\langle\mathtt{p},\mathtt{k}\rangle \ast \mathtt{res}::\mathtt{ll}\langle\mathtt{r}\rangle \wedge \mathtt{p}=\mathtt{res} \wedge \mathtt{k}=\mathtt{n} \wedge \mathtt{m}=\mathtt{n}+\mathtt{r}$
		$\mathtt{a}::\mathtt{ll}\langle\mathtt{b}\rangle \ast\!\!\rightarrow \mathtt{a}::\mathtt{ls}\langle\mathtt{c}\rangle \ast \mathtt{res}::\mathtt{ll}\langle\mathtt{d}\rangle \wedge \mathtt{b}=\mathtt{c}+\mathtt{d} \wedge \mathtt{c}\leq\mathtt{n}$
	2.322	$\mathtt{x}::\mathtt{sllB}\langle\mathtt{S}\rangle \wedge \mathtt{n}\geq 0\wedge\lvert\mathtt{S}\rvert\geq\mathtt{n} \ast\!\!\rightarrow \mathtt{x}::\mathtt{slsB}\langle\mathtt{p},\mathtt{T}\rangle \ast \mathtt{res}::\mathtt{sllB}\langle\mathtt{S}_2\rangle \wedge \mathtt{p}=\mathtt{res} \wedge \lvert\mathtt{T}\rvert=\mathtt{n} \wedge \mathtt{S}=\mathtt{T}\sqcup\mathtt{S}_2 \wedge \forall\mathtt{u}\in\mathtt{T},\mathtt{v}\in\mathtt{S}_2\cdot\mathtt{u}\leq\mathtt{v}$
		$\mathtt{a}::\mathtt{sllB}\langle\mathtt{A}\rangle \ast\!\!\rightarrow \mathtt{a}::\mathtt{slsB}\langle\mathtt{A}_1\rangle \ast \mathtt{res}::\mathtt{sllB}\langle\mathtt{R}\rangle \wedge \mathtt{A}=\mathtt{A}_1\sqcup\mathtt{R} \wedge \lvert\mathtt{A}_1\rvert\leq\mathtt{n} \wedge \forall\mathtt{b}\in\mathtt{A}_1,\mathtt{c}\in\mathtt{R}\cdot\mathtt{b}\leq\mathtt{c}$
Binary tree, binary search tree, AVL tree and red-black tree processing programs		
height	0.821	$\mathtt{x}::\mathtt{bt}\langle\mathtt{S},\mathtt{h}\rangle \ast\!\!\rightarrow \mathtt{x}::\mathtt{bt}\langle\mathtt{T},\mathtt{k}\rangle \wedge \mathtt{res}=\mathtt{h}=\mathtt{k} \wedge \mathtt{S}=\mathtt{T}$
		$\mathtt{a}::\mathtt{bt}\langle\mathtt{A},\mathtt{b}\rangle \wedge \mathtt{a}\neq\mathtt{null} \ast\!\!\rightarrow \mathtt{a}::\mathtt{node2}\langle\mathtt{c},\mathtt{d},\mathtt{e}\rangle \ast \mathtt{d}::\mathtt{bt}\langle\mathtt{D},\mathtt{f}\rangle \ast \mathtt{e}::\mathtt{bt}\langle\mathtt{E},\mathtt{g}\rangle \wedge$ $\mathtt{A}=\{\mathtt{c}\}\sqcup\mathtt{D}\sqcup\mathtt{E} \wedge \mathtt{b}=\mathtt{max}(\mathtt{f},\mathtt{g})+1 \wedge (\mathtt{res}=\mathtt{d} \vee \mathtt{res}=\mathtt{e})$
search	1.851	$\mathtt{x}::\mathtt{bst}\langle\mathtt{sm},\mathtt{lg}\rangle \ast\!\!\rightarrow \mathtt{x}::\mathtt{bst}\langle\mathtt{mn},\mathtt{mx}\rangle \wedge \mathtt{sm}=\mathtt{mn} \wedge \mathtt{lg}=\mathtt{mx} \wedge 0\leq\mathtt{res}\leq 1$
		$\mathtt{a}::\mathtt{bst}\langle\mathtt{b},\mathtt{c}\rangle\wedge\mathtt{a}\neq\mathtt{null} \ast\!\!\rightarrow \mathtt{a}::\mathtt{node2}\langle\mathtt{d},\mathtt{e},\mathtt{f}\rangle\ast\mathtt{e}::\mathtt{bst}\langle\mathtt{b},\mathtt{g}\rangle\ast\mathtt{f}::\mathtt{h}\langle\mathtt{c}\rangle\wedge\mathtt{g}\leq\mathtt{d}\leq\mathtt{h}$
avl_ins	5.202	$\mathtt{x}::\mathtt{avl}\langle\mathtt{S},\mathtt{h}\rangle \ast\!\!\rightarrow \mathtt{res}::\mathtt{avl}\langle\mathtt{T},\mathtt{k}\rangle \wedge \mathtt{T}=\mathtt{S}\sqcup\{\mathtt{v}\} \wedge \mathtt{h}\leq\mathtt{k}\leq\mathtt{h}+1$
		$\mathtt{a}::\mathtt{avl}\langle\mathtt{A},\mathtt{b}\rangle \ast\!\!\rightarrow \mathtt{a}::\mathtt{avl}\langle\mathtt{A},\mathtt{b}\rangle \wedge \mathtt{res}=\mathtt{b}$
rbt_ins	9.093	$\mathtt{x}::\mathtt{rbt}\langle\mathtt{S},\mathtt{cl},\mathtt{bh}\rangle \ast\!\!\rightarrow \mathtt{res}::\mathtt{rbt}\langle\mathtt{T},\mathtt{cl}_1,\mathtt{bh}_1\rangle \wedge \mathtt{T}=\mathtt{S}\sqcup\{\mathtt{v}\}$
		$\mathtt{a}::\mathtt{rbt}\langle\mathtt{A},\mathtt{b},\mathtt{c}\rangle \ast\!\!\rightarrow \mathtt{a}::\mathtt{rbt}\langle\mathtt{A},\mathtt{b},\mathtt{c}\rangle \wedge \mathtt{res}=\mathtt{b}$

Table 2. Selected experimental results (sorting)

Prog.	**Time**	**Main spec.** ($\Phi_{pr} \ast\!\!\rightarrow \Phi_{po}$) or **Derived unknown spec.** ($\Phi^u_{pr} \ast\!\!\rightarrow \Phi^u_{po}$)
Sorting (main)		$\mathtt{x::llB\langle S\rangle} \ast\!\!\rightarrow \mathtt{res::sllB\langle T\rangle} \wedge \mathtt{T{=}S}$
merge	4.099	$\mathtt{a::sllB\langle A\rangle} * \mathtt{b::sllB\langle B\rangle} \ast\!\!\rightarrow \mathtt{res::sllB\langle R\rangle} \wedge \mathtt{R{=}A{\sqcup}B}$
quick	2.064	$\mathtt{a::lbd\langle A\rangle} \ast\!\!\rightarrow \mathtt{a::lbd\langle A_1\rangle} * \mathtt{res::lbd\langle R\rangle} \wedge \mathtt{A{=}A_1{\sqcup}R} \wedge \forall \mathtt{c}{\in}\mathtt{A_1}, \mathtt{d}{\in}\mathtt{R} \cdot \mathtt{c}{\leq}\mathtt{b}{\leq}\mathtt{d}$
unknown	1.824	$\mathtt{a::llB\langle A\rangle}{\wedge}\mathtt{a}{\neq}\mathtt{null} \ast\!\!\rightarrow \mathtt{res::node\langle c,b\rangle}{*}\mathtt{b::llB\langle B\rangle}{\wedge}\mathtt{A}{=}\{\mathtt{c}\}{\sqcup}\mathtt{B}{\wedge}\forall\mathtt{d}{\in}\mathtt{B}{\cdot}\mathtt{c}{\leq}\mathtt{d}$

It can be seen that all programs are successfully verified, with some obligations on the unknown calls discovered. We note down two observations on the experimental results. The first is that the discovered specifications for the unknown procedures are usually more general than what we expect. Bear in mind that we have replaced some instructions from those programs with unknown calls. We have compared the inferred specifications for those unknown calls with the original instructions. The results show that the specifications derived by our algorithm not only fully capture the behaviours of those instructions, but also suggest other possible implementations. A case in point is list's `travrs`. Its "unknown call" was originally an assignment `x = x.next` which traverses the list towards its end by one node. We are able to infer that the unknown call may actually traverse the list for arbitrary number of nodes, provided it does not go beyond the list's tail or where the user has specified as input, which allows more implementations for the unknown procedure to be verified.

The second observation is that the precision of unknown calls' discovered specifications depends on its caller's given specification. As can be seen we have verified several list-processing programs where each one has various specifications. Within these programs we want to point out that the ones with specifications of both normal lists and sorted lists share the same code (but just with two different specifications). Such examples include `create`, `sort_insert`, `delete`, and so on. For `create` which creates a list containing numbers from 1 to `n` in descending order, we can see once incorporated with `llB` as specification predicates, the unknown call is expected to return a node whose value `c` is within 1 to `n`. Comparatively, when verified for sortedness, `c` is inferred to be between `a−1` and `a`, as for sortedness to hold. For `delete`'s sorted version, we also have the extra information that the list with one node removed is still a sorted list (with the multi-set value constraints), whose result is stronger than the normal list version.

7 Conclusion

It is a practical and challenging problem to verify the full functional correctness of heap-manipulating imperative programs with unknown procedure calls. Our proposed solution infers expected specifications for unknown procedures from their calling contexts. The program is verified correct on condition that the invoked unknown procedures meet the inferred specifications. We employ a

forward program analysis over a combined domain and invent a novel abduction for it to synthesise the specifications of the unknown procedure. As a proof of concept, we have also implemented a prototype system to test the viability of the proposed approach. Our main future work is to explore more general solution for unknown calls in sequence to achieve more reasonable specifications for them.

Acknowledgement. This work was supported in part by the EPSRC projects EP/G042322/1 and EP/E021948/1.

References

1. Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: POPL (2002)
2. Beizer, B., Wiley, J.: Black-box testing: techniques for functional testing of software and systems. IEEE Software 13(5) (September 1996)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: 36th POPL (January 2009)
4. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: PLDI (1994)
5. Giacobazzi, R.: Abductive analysis of modular logic programs. In: ILPS (1994)
6. Gopan, D., Reps, T.: Low-level library analysis and summarization. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 68–81. Springer, Heidelberg (2007)
7. Kozaczynski, W., Booch, G.: Component-based software engineering. IEEE Software 15(5), 34–36 (1998)
8. Luo, C., Craciun, F., Qin, S., He, G., Chin, W.-N.: Verifying pointer safety for programs with unknown calls. Journal of Symbolic Computation (to appear)
9. Qin, S., Luo, C., He, G., Craciun, F., Chin, W.-N.: Verifying heap-manipulating programs with unknown calls. Research report, Teesside University (2010), http://www.scm.tees.ac.uk/s.qin/papers/unknown.pdf
10. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Thor: A tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
11. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
12. O'Hearn, P.W., Yang, H., Reynolds, J.C.: Separation and information hiding. In: 31st POPL (January 2004)
13. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17th LICS (2002)
14. Sessions, R.: COM and DCOM: Microsoft's vision for distributed objects. John Wiley & Sons, Inc. New York (1998)
15. Szyperski, C.: Component technology: what, where, and how? In: ICSE (2003)
16. Woodcock, J.: Verified software grand challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 617–617. Springer, Heidelberg (2006)

API Conformance Verification for Java Programs

Xin Li, H. James Hoover, and Piotr Rudnicki

Dept. of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E8
{xinli,hoover,piotr}@cs.ualberta.ca

Abstract. Software components, services, or modules are used via their application programming interface (API). For any sufficiently complex component, there are strict rules on the order and context in which particular methods of the API can be invoked. For example, a file must be opened before reading, and not read after closing. These constraints are called API conformance rules. Their violation at run-time creates errors, which are often subtle and difficult to diagnose. In general, API conformance rules cannot be statically checked if concurrency is involved. We present a verification framework, called Fex, that assists in Java API conformance verification. Fex operates as follows. The first step is to express the API conformance rules as executable specifications. Then, the program under investigation is instrumented such that all potential exceptions can be easily raised. Next, the program is sliced to retain only control flow and the relevant APIs. The executable API conformance rules and sliced program are then processed by the Java Path Finder model checker. Possible violations of the conformance rules are exhibited as exceptions during model checking. We have successfully applied our framework to the TSAFE reference air traffic control system and identified a subtle deadlock missed by previous verification efforts.

Keywords: API conformance, verification, model checking, exception.

1 Introduction

Component-based software development is a widely used design approach in software engineering. In general, a large system can be decomposed into several functional components. Every component has a well-defined application programming interface (API) which is used for communicating across the components. The API implementers only need to focus on implementing the services that the API promises. As a result, these components can then be reused for other applications, thus improving development efficiency and quality.

The API of a software component declares its methods together with types of parameters and types of results. Whether a method has been passed appropriate-type parameters is checked at compilation time. However, a non-trivial API will impose additional constraints on the order of calling of its methods. For example,

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 188–203, 2010.
© Springer-Verlag Berlin Heidelberg 2010

a network communication application using the `Socket` API is expected to follow constraints like these:

– Operations such as `getInputStream` or `getOutputStream` can only be applied on a `Socket` object which has already connected to a server.
– After the operation `close`, there should not be any further operations performed on this `Socket` object.

Constraints of this type are called *API conformance rules*. In general, such rules cannot be checked statically if concurrency is involved. Violations of the rules are raised as run-time exceptions.

In this paper, we present a verification framework named Fex, which combines static analysis and model checking techniques to verify program's API conformance. The original program is instrumented with support for exploring all possible flows of control including the exceptional ones so that the back-end model checker is then able to inspect all possible states of the modified program. To avoid the state explosion problem, we use program slicing that attempts to preserve the complete control flow in the face of environmental and data diversity.

In Section 2 we use a simple example to illustrate an API conformance problem and how the Fex tool is used to verify it. In Section 3, we discuss our framework for API conformance verification. A bigger case study is given in Section 4.

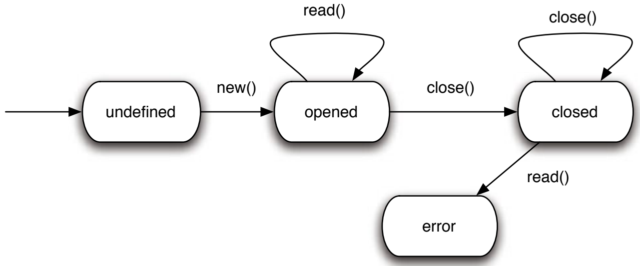


Fig. 1. An FSM specification of class `FileInputStream`

2 A Motivating Example

The event sequence constraints described by the Finite State Machine (FSM) of Figure 1 presents a simplified API conformance specification for Java class `java.io.FileInputStream`.[1] We would like to statically verify that a given program does not violate `FileInputStream` conformance rules. Instead of running the actual program, we use the JPF model checker which examines all execution paths. The idea is to replace the concrete, native coded implementation

[1] Note that our specification allows a close operation after the stream is closed which is exactly what the real implementation does.

of `FileInputStream` with an executable specification of `FileInputStream` conformance rules. This executable specification of conformance rules is a much simpler program than the actual implementation which implicitly encodes the rules.

Figure 2 presents the executable specification of `java.io.FileInputStream` implementing the FSM from Figure 1. The FSM state is represented by the field `status`. Transition relations are coded as `if` statements. For example, the `if` statement inside method `close` changes the program state from `opened` to `closed`. When the FSM enters the `error` state, an exception is raised in the program, representing a possible conformance violation.

```
package java.io;
public class FileInputStream extends InputStream {

  private enum State {undefined, opened, closed}
  public State status = State.undefined;
  public final int CONSTANT_INTEGER = 2;

  public FileInputStream(File file) {
    if (status == State.undefined)  status = State.opened;
  }
  public FileInputStream(String s) {
    if (status == State.undefined)  status = State.opened;
  }
  public int read() {
    if (status == State.closed)
       throw new Error("Conformance Error! Read after stream closed.");
    if (status == State.opened)  status = State.opened;
    return CONSTANT_INTEGER;
  }
  public int read(byte[] b) {
    if (status == State.closed)
       throw new Error("Conformance Error! Read after stream closed.");
    if (status == State.opened)  status = State.opened;
    return CONSTANT_INTEGER;
  }
  public void close() {
    if (status == State.opened) status = State.closed;
    }
    // ... ...
}
```

Fig. 2. Executable specification for Class `java.io.FileInputStream`

We would like to verify that the very simple program in Figure 3 obeys the conformance rules of `FileInputStream`. The program first calls the method `initialize` which creates a `FileInputStream` object, named `is`, connected to file named `args[0]`, and a `FileOutputStream` object, named `os`, connected to file

```
04 class Test {
05     private FileInputStream is;
06     private FileOutputStream os;
07     void initialize(String s1, String s2) {
08         try{
09             is = new FileInputStream(s1);
10             os = new FileOutputStream(s2);
11         }
12         catch (IOException e) {
13             System.out.println("Catching Exceptions, now clean up.");
14             cleanUp();
15         }
16     }
17     void copy () {
18         try{
19             int i = is.read();
20             while(i != -1) {
21                 os.write(i);
22                 i = is.read();
23             }
24         }
25         catch (IOException e) {
26             System.out.println("Catching Exceptions, now clean up.");
27             cleanUp();
28         }
29     }
30     void cleanUp() {
31         try{
32             if (is != null) is.close();
33             if (os != null) os.close();
34         }
35         catch (IOException e) {
36             // do something ...
37         }
38     }
39     public static void main (String[] args) {
40         Test t = new Test();
41         t.initialize(args[0], args[1]);
42         t.copy();
43     }
44 }
```

Fig. 3. Example program using `java.io.FileInputStream`

named `args[1]`. The method `copy` copies the contents from `is` to `os`. Method `cleanUp` is called when an exception situation is raised.

The original program in Figure 3 is converted into a sliced Java program as in Figure 4.[2] We perform two actions during the conversion.

- We instrument the program so that the program can non-deterministically raise all exceptions we want to examine.
- We slice the program in order to make the possible program states smaller.

Both actions are controlled by a configuration file that specifies what kind of exceptions should be instrumented and which APIs are considered relevant and should be preserved. For this example, as the corresponding configuration file marks both `FileInputStream` and `FileOutputStream` as relevant, objects `is` and `os` with corresponding operations are preserved. Non-relevant APIs are removed to minimize the search space, for example the statement `System.out.println`. The configuration file (see [8] for technical details) defines the abstraction function which is used to guide the program instrumentation and slicing and is manually prepared.

At the model checking phase, the model checker examines the sliced program together with the executable specification for API `java.io.FileInputStream` (Figure 2). If there is a conformance rule violation, the program will raise an exception which is thrown by our executable specification. The back-end model checker can catch this exception and generate a report with the stack state and step by step execution trace leading to the conformance error.

Upon examining the execution trace (produced by JPF but not included here) we found that an `FileNotFoundException` was thrown at line 12 (of the sliced program) and was caught at line 14. Method `cleanUp` was called in turn at line 14 which closed the input stream `is`. Then, method `copy` (line 45) was executed which attempted to call operation `read` (line 18) on closed stream `is`. This violates the API conformance rule of `FileInputStream`.

This error is triggered by calling the `cleanUp` too early inside `initialize`. Usually the clean up method is called at the end of the whole action (preferably in a `finally` block) so that no side effects are left. However, in this example, trying to do the cleaning in method `initialize` results in referencing a closed stream `is`. For this example, the proper approach would be to add a `try-finally` block in method `main` and call `cleanUp` in the `finally` section.

This example also illustartes another API conformance error. In the state `undefined`, the only legal operation is to execute a class constructor. Attempting any other operation results in a null pointer reference. The above program can lead to this situation. If operation `new FileInputStream()` on line 09 fails, variable `is` is null and an `IOException` is raised. Continuing to execute `cleanUp` at line 14 is OK, but executing method `copy` which attempts operation `read` on a null object `is` raises `NullPointerException`. Fex catches this error as well.

[2] This automatically generated sliced program has been edited due to space constraints.

```
04 class Test {
05    private FileInputStream is;
06    private FileOutputStream os;
07    void initialize(String s1, String s2) {
08     try{
09     if (Verify.getBoolean()) is = new FileInputStream((String)null);
10     else throw new FileNotFoundException();
11     if (Verify.getBoolean()) os = new FileOutputStream((String)null);
12     else throw new FileNotFoundException();
13     }
14     catch (IOException e) { this.cleanUp(); }
15    }
16    void copy() {
17      try {
18          if (Verify.getBoolean()) is.read();
19          else throw new IOException();
20          for (int JPF_index0 = 0; JPF_index0 < 2; JPF_index0++){
21              if (Verify.getBoolean()) this.os.write(2);
22              else throw new IOException();
23              if (Verify.getBoolean()) this.is.read();
24              else throw new IOException();
25          }
26      }
27      catch (IOException e) { this.cleanUp(); }
28    }
29    void cleanUp() {
30      try {
31          if (is != null) {
32             is.close();
33             if (Verify.getBoolean()) throw new IOException();
34          }
35          if (os != null) {
36             os.close();
37             if (Verify.getBoolean()) throw new IOException();
38          }
39      }
40      catch (IOException e) { }
41    }
42    public static void main(String[] args) {
43        Test t = new Test();
44        t.initialize((String)null, (String)null);
45        t.copy();
46    }
47 }
```

Fig. 4. Sliced program based on Figure 3

3 Fex: A Model Checking Framework for API Conformance Verification

Verification of control intensive properties like API conformance is challenging for both human effort and machine resources. Efficient static analysis techniques only inspect the code at compile time; they are quite imprecise as they only approximate the inter-procedural control-flow. Model checking can explore all possible run-time behaviors, but of a greatly simplified program. In practice these two approaches work together to increase our confidence.

3.1 Specifying Conformance Rules with Executable Specification

API conformance rules specify the order in which API methods may be called. These rules are implicitly encoded inside the actual API implementation. Run-time exceptions are raised when these conformance rules are violated. The description of these rules are written as comments to the API and are organized in an informal and unstructured way. In order to do the API conformance verification, explicit specification of these rules is required. We do so in an executable specification.

Previous efforts [5,7,12] on formally specifying API conformance rules are based on FSM. While FSMs are adequate to describe simple behavior inside a single API, they are insufficient for handling complex API behaviors which might involve potentially unbounded numbers of cases. For example, Java API class `java.util.concurrent.ReentranReadWriteLock` is used to control concurrent read-write issues. Because this lock class is re-entrant, it may involve an unbounded number of lock/unlock operations. A FSM cannot describe nested behaviors of `ReentrantReadWriteLock` such as locking an unbounded number times.

We are inspired by [11] to use *executable specifications* to specify the API conformance rules. Our executable specifications are written in Java. Each API under investigation needs to have an executable specification Java class. If the real implementation of this API class only involves pure Java and the possible program states are manageable, it can be used as its own executable specification. However, for the APIs which involve native code or many possible program states, we have to implement a new version of the API. This new implementation only partially implements the API in as much detail as required to capture the conformance rules. For example, as `java.io.FileInputStream` involves native code, we need to implement our version of `java.io.FileInputStream` as the executable specification, see Figure 2. This of course means that other errors may be introduced as a result of the partial implementation. For example, file content might not actually be read. Executable specifications give us a chance to mimic the essence of the API's functionality which we want to verify.

An executable specification is more expressive than a pure FSM specification. FSM based specification can be translated into executable specification. Furthermore, by adopting full Java as the specification language, we can specify all kinds of API behaviors. For example, the above `ReentrantReadWriteLock` challenge can be described by introducing a lock counter into the executable specification.

3.2 Verification Process

In Fex, the verification process is divided into three steps (Figure 5): static analysis and code instrumentation, program slicing, and model checking. Each of these three steps is controlled by a configuration file.

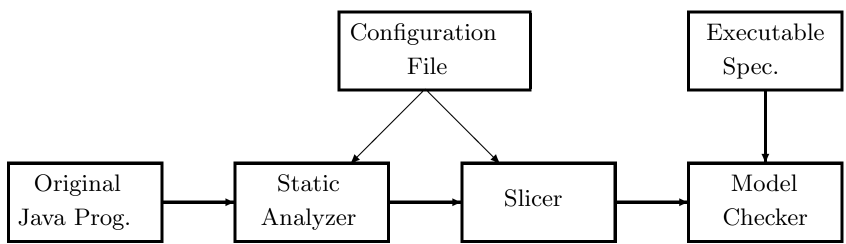


Fig. 5. Verification Process

Program Instrumentation. API conformance violations often arise under situations that are not anticipated or rarely encountered. These situations are frequently related to handling exceptional situations. Therefore, the ability to examine all possible exceptional control flow paths is essential in API conformance verification and thus our framework builds upon our exception safety verification presented in [9].

The original Java program is instrumented such that all possible exceptions can be raised. For each method call, the analyzer first determines if the method source code is available. If available, the analyzer instruments the method directly. If not, the analyzer instruments the exception interface extracted from the byte code of that method. Our static analysis is built on top of the exception analysis tool Jex [13]. Note, that we do not need inter-procedural analysis to determine the control flow graph (Jex does) since actual control flow is examined at the model checking phase.

Expressions or statements which might throw an exception are instrumented as follows:

```
if (Verify.getBoolean()) statement; else throw new ExceptionType();
```

`Verify` is a special class provided by the back-end model checker. This class is used to model non-deterministic choice. Calling `getBoolean()` from class `Verify` represents a branch point in the control flow. During state exploration the method returns a boolean value non-deterministically, thus causing the model checker to explore both paths.

The instrumentation step is guided by a configuration file that states which exceptions are considered essential and need to be instrumented and which program fragments are considered crucial and need to be preserved. For details about the role of the configuration file, please see [8,9].

Program Slicing. Model checking of production Java programs in full detail is impractical because of state explosion. Since we are interested in properties that are dominated by control flow, we use rather aggressive program slicing [16] to transform the original Java program into one with a substantially smaller state space while hoping to preserve the interesting control flow properties.

In Fex, the instrumented program from step 1 is fed into a slicer. The sliced program contains nothing more but control flow constructs and all components of the API which we are verifying. For example, Figure 4 presents the sliced program whose original program is in Figure 3.

We slice every Java program under investigation such that

1. Both branches of a conditional statement will be executed.
2. All loop constructs are replaced by a fixed loop iteration which executes each loop at most twice, as a result some errors can be missed or introduced.
3. Variable types are divided into four categories
   (a) Primary Java types (*PrimType*).
   (b) Control-flow dependent types, e.g. the `Thread` class.
   (c) Crucial types (PType) are types related to API conformance checking, e.g. the API classes that are under investigation.
   (d) Ignored types (LType) are types that come from the Java library or from third party application whose source code is not available. Since we do not know how the classes in LType behave we assume that they obey the conformance rules. Such classes are trusted and ignored in further verification.

   Class fields of primary or ignored type are removed. Methods whose receiver's type is an ignored type are removed. Parameters of primary or ignored type are replaced with a fixed value, e.g. `NULL` for reference types, `2` for integer, etc.

The above slicing criteria can be described in the format of program transformation rules. We have 15 transformation rules in total covering all essential Java program constructs. Here, we list the most interesting one. For the complete list of rules, please see [8,9]. (The function `type(`*expr*`)` is used to return the type of *expr*.)

$$[\mathit{Stmt}]\ \ \mathtt{type}(E_1) \notin \text{LType}\ \ \&\ \mathtt{type}(\mathit{expr}) \in \text{LType} \qquad \frac{E_1.\text{MName}(\mathit{expr},\ E^*)}{E_1.\text{MName}(\mathtt{null}, E^*)}$$

This rule states that if we invoke a method MName via an object E_1 whose type is not to be ignored, we can still eliminate the parameters that are supposed to be ignored by replacing these parameters with a fixed value. This rule preserves the method call while projecting away the parameters that are not directly involved in API conformance check, although their values may in fact be important.

Model checking. In the model checking phase, We use the software model checker, Java Pathfinder (JPF) [15] as our back-end model checker. Based on

the sliced program and the executable specification, JPF explores all possible execution paths. As API conformance rules are expressed in executable specifications which are directly fed into JPF without any abstraction, JPF can catch any conformance violations in the sliced program. Upon detecting a violation, JPF dumps out an execution path leading to the error. The execution path can help the programmer to fix the conformance violation problem. This verification process is iterated until no further violation is reported.

4 Experimental Results

We now present the experimental results obtained by applying Fex to verify conformance rules of Java concurrency API in a Air Traffic Control System called TSAFE. Fex has also been successfully applied on several other case studies whose focus were on Java Socket API, Java IO stream API, Java collection APIs and JDBC API.

4.1 Introduction to TSAFE

The main goal of Automated Air Traffic Control (AATC) [10] is to reduce the load on air traffic controllers as they monitor and direct aircraft. TSAFE, which stands for The Tactical Separation Assisted Flight Environment, is the most important component of the AATC system. It is designed to help air traffic controllers in detecting and resolving short-term conflicts between aircrafts. Currently, the traffic controllers maintain the aircraft separation by monitoring the radar data for possible conflicts and give instructions to pilots to maneuver to avoid near-collision. Automating this conflict avoidance process is the essential part of the whole AATC process and the result should be highly dependable, more so than for other components.

A prototype of TSAFE was implemented at MIT [6]. Based on this original reference implementation, researchers from the Fraunhofer Center-Maryland (FC-MD) have developed TSAFE based testbed set[10]. This set contains several version of TSAFE, each version contains artifacts including design documentations, program specifications, source codes and programs with seeded faults. As part of NASA's High Dependability Computing Project (HDCP), the purpose of this testbed set is to provide a platform to evaluate a wide range of new technologies for improving the dependability of mission-critical systems.

As our testbed, we use TSAFE III, a distributed client-server TSAFE version which is provided by FC-MD. The TSAFE III implementation consists about 90 classes and 21,000 lines of Java code. The overview of the software architecture is presented in Figure 6. The trajectories of the flights are stored in the flight database. The database is periodically updated by the radar feed thread through TCP/IP protocol. The computation component is used to do the flight conformance checking. The client side implements GUI based display functionality. It communicates to the server via a RMI protocol. Multiple clients are allowed to communicate to the server at the same time.

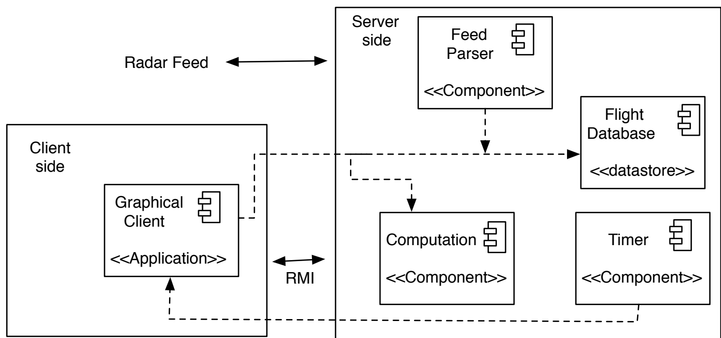


Fig. 6. TSAFE III Architecture

In the original TSAFE III implementation, the flight database manipulations are coded using the Java synchronization mechanism to avoid data corruption. All database read/write operations are synchronized so that there can only be one database access operation at a given time. This implementation is not very efficient because read operations cannot be done concurrently.

Betin-Can et al. [2] proposed to use a read-write lock to solve this problem. A read-write lock guarantees that multiple readers can access the flight database at the same time, but a writer can only access the database exclusively. They then re-engineered the original TSAFE into a version which adopts a customized read-write lock solution. When verifying the correct usage of the read-write lock under TSAFE III, they decouple the behavior of the read-write lock from the threads that use them by replacing the read-write lock with a simplified Finite State Machine(FSM) specification.

The approach of [2] is a good first step, however it has three weaknesses:

1. Java 5 provides a new API (`java.util.concurrent`) to handle all concurrent control related issues. Class `java.util.concurrent.ReentrantReadWriteLock` is designed specifically for read-write lock usage. This API provides the same functionality as the custom read-write lock. Thus it makes sense, on the grounds of component reuse, to use this standard Java API rather than a customized read-write lock API.
2. During the verification, a FSM specification is used to replace the real read-write lock code. This FSM specification is not the real code but a simplified version.
3. The original verification only handles a single threaded scenario. However, there are many errors which only appear under a multi-threaded scenario. For example, lock starvation scenario, where one thread holds the lock and never releases it.

Our case study addressed these issues. We re-engineered the existing code by replacing the custom read-write lock and associated operations with the standard Java concurrency lock API `ReentrantReadWriteLock` and associated

operations. Because the custom read-write lock API is a subset of the standard Java read-write lock API, this re-engineering process guarantees that if there is any `ReentrantReadWriteLock` related API conformance error, that error also exists in the prior implementation.

Because the implementation of the `ReentrantReadWriteLock` API is written in pure Java, we use the original implementation as our executable specification. Thus, we directly verify the real API implementation and avoid a FSM.

Finally, the aggressive program abstraction of Fex reduces the state space sufficiently that we can then handle the multi-thread scenario.

According to Java API specification [1], the conformance rules for this reentrant read-write lock are:

- When using the `ReentrantReadWriteLock`, a lock instance should first be initialized. Then, a read lock or write lock can be acquired from this instance and the corresponding lock/unlock operation can be conducted upon the acquired read/write lock.
- It is legal for both readers and writers to re-acquire read or write locks, but the write lock process can only be done exclusively. This means read operations are not allowed until all write locks held by the writing thread have been released.
- A lock operation should always be followed with a corresponding unlock operation in the near future otherwise a deadlock situation may occur.
- The lock/unlock operation is re-entrant. So multiple lock operations should be followed with the same number of unlock operations.
- A write lock can be downgraded to a read lock. That means when a write lock is held, a read operation can be executed. However, upgrading from a read lock to the write lock is not allowed.

4.2 Experiment Results and Discussion

As TSAFE III is based on the distributed client-server architecture and there are several possible entry points for the application. In order to cover the whole application, a proper test harness (collection of main programs) is needed as JPF needs an entry point.

Our test harness contains several test cases. Each test case covers one possible thread combination. These test cases are coded following the typical TSAFE behavior patterns. For example, a client thread will first try to connect with a server via RMI, and then read the information from the back-end database.

In reality, TSAFE is used in a concurrent environment, with multiple clients and radar feed processes. But even with the aggressive slicing of Fex, we can only verify at most three threads for the TSAFE system with the real Java read-write lock implementation.

However, this was sufficient to expose a subtle error when model checking the multi-thread scenario with a radar feed parser thread. Inside the `ASDIParser` class, a method `extractMessage` from class `MessageExtractor` was called to extract information from the database. This method call is protected by an instance

of `ReentrantReadWriteLock`. The programmer's intention is to first execute the `lock` operation, then the `extractMessage` method and finally execute the `unlock` operation. At the first glance, this should work properly since the `lock` and `unlock` operations are paired.

However, the code is flawed. When executing the `extractMessage` operation, there is the possibility that it throws a runtime exception. For example, if the incoming message is damaged with corrupted latitude information, the `extractMessage` procedure still invokes the `getLatitude` method from class `NASFields`, and a `NumberFormatException` is raised inside `getLatitude` method and propagates.

Also, for any unrecognized message, the `extractMessage` method itself will respond with an `RuntimeException`. All these potential unhandled exceptions will cause the program to bypass the associated `unlock` operation. This is actually the case reported by Fex. Since the feed parser works as a demon thread which updates the flight database periodically, the bypassed `unlock` operation may cause the client accessing thread to deadlock.

This error is triggered by neglecting unhandled exceptions. The correct way to do it is to always put the operations between `lock` and `unlock` into a `try` block and put the `unlock` operation into the corresponding `finally` block. In that way, under all circumstances, the `unlock` operation will eventually be executed. This potential deadlock situation also exists in [2]'s implementation but because their verification framework cannot handle multi-threaded scenarios and the possible `NumberFormatException` is not in their verification scope, it was not detected.

Table 1. Running time and state space size for the experiments

	Verify		Falsify	
Thread Info	Visited States	Time(sec)	Visited States	Time(sec)
1 CT	2696	26	-	-
1 FT	3090	26	-	-
1 CT, 1 FT	1011295	490	33	25
2 CT, 1 FT	38636247	19315	38	26

Table 1 demonstrates the performance of the JPF model checker while verifying the sliced TSAFE system.[3] In this table, CT means Client Thread, FT means Radar Feed Thread. The whole performance table is divided into two parts: Falsify and Verify. Under the Falsify column, the visited states and corresponding execution time for finding the deadlock error are listed. Under the Verify column, the visited states and corresponding execution time for verifying

[3] All experiments were performed on a Mac Book Pro with Intel Core 2 Duo, 2.26 GHz, 2.0 GB RAM, running Mac OS X 10.5.8, Sun Java SDK build 1.5.0_24-149 and Java Pathfinder Version 3.1.2.

a error-free TSAFE version is listed. As we can see from the table, under the one thread scenario, we cannot detect the deadlock because it only shows up under a multi-threaded scenario.

It is worth noting that after fixing the error, we attempted another verification under 2 client threads and 1 radar feed thread. After running almost 6 hours and visiting approximately 40 million program states, JPF exhausted heap memory and further exploration terminated. Thus other errors may be present.

In order to further explore the effectiveness of the Fex tool, we have applied Fex on 52 fault seeded TSAFE versions. Upon the construction of these fault seeded version, we follow the idea from Betin-Can et al. [2] and compared our results with theirs. They categorize all seeded faults into two groups. The first group is called **modified-call faults** which includes adding, removing or swapping lock related operations. The second group is called **conditional-call faults** which includes adding a branch condition in front of these lock related operations. During the verification process, batches of slight variations of the program are generated by seeding with program faults from these categories.

Usually for each batch, only one program fault is seeded. The effectiveness of the verification framework can be evaluated by the numbers of program faults revealed among all batches.

In our experiment, There are in total 16 lock/unlock operations scattered around the whole application. For modified-call faults, every related operation will be added, removed or swapped once. For conditional-call faults, a fault is seeded by adding a branch whose guard is based on an independent incrementing program variable. For example, a conditional statement in the form of : `if ( i > 100) statement;`. In order to exhaust all possible fault styles for all relevant operations, we prepared 53 versions of TSAFE (one of which is an unseeded version).

We ran the experiment in 53 batches. In each batch, a TSAFE version is randomly picked. Without knowing the details about the seeded fault's category (it can also be the unseeded version as well) and the exact place of the seeded fault, Fex is used in the verification process. Fex found all seeded faults from all batches and identified the unseeded version. This compares favourably to Betin-Can's verification approach, which can resolve most of the seeded faults with the exception of data dependent faults. These deep embedded faults are challenging to discover as we would need to consider a very large number of program states. Generally, these deep embedded faults cannot be model checked without some substantial data abstraction.

5 Related Work

There are several projects that focus on API conformance verification. The projects which are closest to our work are Fugue [5] and Plural [3,4]. Fugue is a static software analyzer based on type-state checking. Type-states [14] specify extra abstract states of objects beyond the usual types. Operations which change object states change their type state as well. Extra annotations are used

to specify API conformance protocols and aliasing information. Therefore, by type-state checking, Fugue can detect possible conformance violations. Bierhoff et al. [4] improves Fugue by introducing *Access Permissions* which extends the type-state refinement ability and applies the updated annotation method to support more expressive specifications and eases aliasing recognition. Based on this improvement, a static analysis tool, called Plural, has been created to conduct Java API conformance checking. Their analysis is based on control flow graphs and currently can only handle single thread scenarios.

Ramalingam et al. [11] use an alternative way to handle API conformance. They specify the API conformance specification with a Java like language `EASL/P`. The original Java program is translated into TVP program (a language based on first order logic to specify operational semantics). By applying a parametric shape analysis on the heap while checking through the program control graphs, the checker can dig out possible conformance violations.

Betin-Can et al. [2] presents *Design for Verification* approach which can be used for API conformance check. Conformance rules are described in FSM. Upon verification, human efforts such as program simplification, program stub preparation, data dependency analysis and thread isolation are needed. So far their approach neglects the influence of program exceptions and can only handle multi-thread scenario by thread isolation.

6 Conclusion

We have applied the Fex tool to the TSAFE safety critical air traffic control system and detected a subtle deadlock error missed by previous verification efforts.

We extend upon previous work [4,2,5,7,12] in several ways.

- Our method uses real Java as the executable specification which can specify properties that a FSM cannot express.
- To the best of our knowledge, we are the first to consider "all" exceptions. The exceptions we want to handle are configurable and we can explicitly ignore some of them.
- We base our analysis on the implicit program model (sliced program) rather than on the program control flow graph.
- We can handle almost all program constructs, including global variables, exceptions and concurrency features.
- For aliasing analysis, since we are monitoring the real VM heap, aliasing can be handled directly without an extra annotation burden.

Acknowledgments

The authors would like to thank Dr. Mikael Lindvall for providing the source code of TSAFE system. We also thank Dr. Aysu Betin-Can for her kind help.

References

1. Java 2 platform api specification, http://download-llnw.oracle.com/javase/1.5.0/docs/api/index.html
2. Betin-Can, A., Bultan, T., Lindvall, M., Lux, B., Topp, S.: Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. Automated Software Engg. 14(2), 129–178 (2007)
3. Bierhoff, K.: Api protocol compliance in object-oriented software, PhD Thesis, Carnegie Mellon University, School of Computer Science (2009)
4. Bierhoff, K., Beckman, N.E., Aldrich, J.: Practical api protocol checking with access permissions. In: Drossopoulou, S. (ed.) ECOOP 2009 – Object-Oriented Programming. LNCS, vol. 5653, pp. 195–219. Springer, Heidelberg (2009)
5. DeLine, R., Fähndrich, M.: Typestates for objects. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 465–490. Springer, Heidelberg (2004)
6. Dennis, G.: Tsafe: Building a trusted computing base for air traffic control software, Master's Thesis, MIT (2003)
7. Fink, S., Yahav, E., Dor, N., Ramalingam, G., Geay, E.: Effective typestate verification in the presence of aliasing. In: ISSTA (2006)
8. Li, X.: Fex: A model checking framework for event sequences, Technical report TR08-14, University of Alberta (2008)
9. Li, X., James Hoover, H., Rudnicki, P.: Towards automatic exception safety verification. In: FM, pp. 396–411 (2006)
10. Lindvall, M., Rus, I., Shull, F., Zelkowitz, M.V., Donzelli, P., Memon, A.M., Basili, V.R., Costa, P., Tvedt, R.T., Hochstein, L., Asgari, S., Ackermann, C., Pech, D.: An evolutionary testbed for software technology evaluation. NASA Journal of Innovations in Systems and Software Engineering 1(1), 3–11 (2005)
11. Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving specialized program analyses for certifying component-client conformance. In: PLDI, pp. 83–94 (2002)
12. Reiss, S.P.: Specifying and checking component usage. In: AADEBUG, pp. 13–22 (2005)
13. Robillard, M.P., Murphy, G.C.: Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. Softw. Eng. Methodol. 12(2), 191–221 (2003)
14. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Softw. Eng. 12(1), 157–171 (1986)
15. Visser, W., Havelund, K., Brat, G., Park, S., Lerda, F.: Model checking programs. Automated Software Engg. 10(2), 203–232 (2003)
16. Weiser, M.: Program slicing. IEEE Trans. Software Eng. 10(4), 352–357 (1984)

Assume-Guarantee Reasoning with Local Specifications

Alessio Lomuscio[1], Ben Strulo[2], Nigel Walker[2], and Peng Wu[3]

[1] Department of Computing, Imperial College London, UK
a.lomuscio@imperial.ac.uk
[2] BT Innovate, Adastral Park, UK
{ben.strulo,nigel.g.walker}@bt.com
[3] State Key Laboratory of Computer Science, Institute of Software,
Chinese Academy of Sciences, China
wp@ios.ac.cn

Abstract. We investigate assume-guarantee reasoning for global specifications consisting of conjunctions of local specifications. We present a sound and complete assume-guarantee rule that permits reasoning about individual modules for local specifications and draws conclusions on global specifications. We illustrate our approach with an example from the field of network congestion control, where different agents are responsible for controlling packet flow across a shared infrastructure. In this context, we derive an assume-guarantee rule for system stability, and show that this rule is valuable to reason about any number of agents, any initial flow configuration, and any topology of bounded degree.

1 Introduction

Assume-Guarantee reasoning [21,12,5] is one of the key techniques to alleviate state explosion in model checking. In a system composed of a number of reactive modules each module can be regarded as interacting with an abstract environment representing the rest of the system. Properties are then verified with the aid of assumptions characterising the environment of each module. General assume-guarantee rules have been proposed for safety and liveness properties over the last decade [11,4,7,8]. However, large assumptions can still cause scalability issues. The motivation of this paper is to investigate possible ways to reduce the size of the assumptions to be identified and to reuse them for compositional verification.

Our starting point is the observation that a module in a system typically reacts directly with relatively few modules in its environment. However, under general assume-guarantee rules, the assumptions generated from a system property do not exploit this neighbourhood dependency. Consequently, assumptions for a module may contain redundant information about parts of the system that the module does not directly interact with. Moreover, any new modules added to the system can contribute with further redundancy in the assumptions.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 204–219, 2010.
© Springer-Verlag Berlin Heidelberg 2010

In this paper we show that, for a system property that can be represented as the conjunction of local specifications on individual modules, these scalability issues can be resolved by generating assumptions from local specifications. Our main contribution is a new presentation of the assume-guarantee rules to permit reasoning about individual modules for local specifications, yet drawing conclusions on properties of the system as a whole.

Firstly, we present a simple assume-guarantee rule $\mathbf{R_1}$ that we prove to be sound for local specifications. Through a counterexample, we show that this simple rule is not complete because it exploits only the direct dependency between modules.

We then extend rule $\mathbf{R_1}$ towards completeness. This leads to a bounded assume-guarantee rule $\mathbf{R}^\pi$ that we prove to be sound and complete for local specifications. It explores the neighbourhood around each module up to the depth π of the dependency closure of the system. We use this rule to propose a bounded assume-guarantee reasoning approach, in which the dependency between modules is exploited incrementally.

We evaluate the approach through a case study of an optimisation-based congestion control system proposed by Kelly and Voice [15]. The optimisation approach allows a distributed solution for network congestion control. The fact that a congestion control system is stable means that each source in the system reaches an equilibrium flow configuration on the routes available to the source. We analyse the stability of the system by reasoning about local stability of its individual sources. The case study shows that an instantiation of rule $\mathbf{R}^\pi$ for system stability can be applied for reasoning about any number of sources, any initial flow configuration, and any topology of bounded degree. To the best of our knowledge, previous work on model checking of networked systems focused on verifying network protocols under given topologies. The assume-guarantee framework developed in this paper supports verification of network-wide objectives *irrespective of the underlying network topologies.*

Related Work. The history of compositional verification of concurrent systems dates back to late 70s and 80s with the pioneering works by Francez and Pnueli [9], Jones [14] and Misra and Chandy [21]. Since then, considerable effort was devoted to studying the soundness of circular assume-guarantee reasoning. Maier [20] showed that compositional circular assume-guarantee rules cannot be both sound and complete. Kupferman and Vardi [16] presented an automata-theoretic approach to model checking assume-guarantee assertions.

More recently, Giannakopoulou, Păsăreanu et al. [11,4,7,10,24] proposed sound and complete non-circular assume-guarantee rules for safety properties, with support of learning based assumption generation. Nam, Alur et al. [22,23] proposed a symbolic approach to learning-based assume-guarantee reasoning. Farzan, Chen et al. [8] extended the assume-guarantee rules to liveness properties, based on the fact that ω-regular languages preserve the essential closure properties of regular languages.

The idea of reasoning about local specifications has appeared in early works on compositional verification [1,13], where sound circular assume-guarantee rules

were proposed for safety properties. This idea is further expanded in this paper to reduce the size of assumptions, and hence to improve the scalability of assume-guarantee reasoning. Moreover, the bounded rule here presented is shown to be sound and complete and applies to liveness properties. Our approach can also be implemented using symbolic representation, and integrated with learning algorithms for automated assumption generation. Additionally, learning-based methodologies can also benefit from our approach by exploiting assumptions over local alphabets, instead of the global alphabet.

The rest of this paper is organised as follows. The simple rule $\mathbf{R_1}$ and the bounded rule $\mathbf{R}^\pi$ are presented in Section 2 and Section 3, respectively. Section 4 illustrates the case study of network congestion control, with the experimental results reported and discussed in Section 5. The conclusions of this work are summarised in Section 6.

2 Assume-Guarantee Reasoning

In this section we first introduce the notion of module in concurrent systems. Then, we present a simple assume-guarantee rule $\mathbf{R_1}$ that permits reasoning about individual modules for local specifications.

Modules. Technically we adopt the basic notion of reactive module [2] to represent concurrent systems that consist of multiple interacting agents. A module is associated with two classes of variables: *state variables* and *input variables*. The former is controlled by the module and thus defines the module's state; the latter is controlled by others that the module reacts directly with.

We assume a domain D of all variables. For a set X of variables, let D^X be the set of all valuation functions on X. For valuation $\rho\colon X \to D$ and $Y \subseteq X$, $\rho\restriction_Y\colon Y \to D$ is the restriction of ρ to Y, i.e., $(\rho\restriction_Y)(x) = \rho(x)$ for any $x \in Y$.

For valuations $\rho_1 : X_1 \to D$ and $\rho_2 : X_2 \to D$, ρ_1 and ρ_2 are *compatible*, denoted $\rho_1 \sim \rho_2$, if $\rho_1(x) = \rho_2(x)$ for any $x \in X_1 \cap X_2$. For compatible valuations ρ_1 and ρ_2, $\rho_1 \cup \rho_2$ is the extension of ρ_1 and ρ_2 to $X_1 \cup X_2$, i.e., $(\rho_1 \cup \rho_2)(x) = \rho_1(x)$ for $x \in X_1 \backslash X_2$, $(\rho_1 \cup \rho_2)(x) = \rho_2(x)$ for $x \in X_2 \backslash X_1$ and $(\rho_1 \cup \rho_2)(x) = \rho_1(x) = \rho_2(x)$ for $x \in X_1 \cap X_2$.

Definition 1 (Module). *A* module *is a tuple $M = (X, I, Q, T, \lambda, q_0)$, where*

- *X is a finite set of state variables controlled by M;*
- *I is a finite set of input variables that module M depends on with $X \cap I = \emptyset$;*
- *Q is a finite set of states;*
- *$\lambda : Q \to D^X$ labels each state $q \in Q$ with a valuation $\lambda(q) : X \to D$;*
- *$T \subseteq Q \times D^I \times Q$ is a transition relation; each transition $(q, \alpha, q') \in T$, denoted $q \xrightarrow{\alpha}_T q'$, means that the state of M evolves from q to q' under input $\alpha : I \to D$;*
- *$q_0 \in Q$ is the initial state.*

An infinite trace σ of module M is an infinite sequence $q_0\alpha_0q_1\alpha_1\ldots$ such that $q_i \xrightarrow{\alpha_i}_T q_{i+1}$ for any $i \geq 0$. Let $inf(\sigma)$ be the set of all the states that are visited infinitely often in σ.

D^X is referred to as the *local* alphabet of module M, where each $\rho \in D^X$ is a valuation on X. An infinite word $w = \rho_0\rho_1\ldots$ on the local alphabet D^X is *derived* by M if there exists an infinite trace $q_0\alpha_0q_1\alpha_1\ldots$ of module M such that $\rho_i = \lambda(q_i)$ for any $i \geq 0$.

D^I is referred to as the *input* alphabet of module M, where each $\alpha \in D^I$ is a valuation on I. An infinite word $\theta = \alpha_0\alpha_1\ldots$ on the input alphabet D^I is *admitted* by M if there exists an infinite trace $q_0\alpha_0q_1\alpha_1\ldots$ such that $q_i \in Q$ for any $i \geq 0$. Let $\mathcal{I}(M)$ be the set of the input words admitted by M. We say that module M is *closed* if $I = \emptyset$.

We define the composition operator for modules. We choose a notion of composition that explicitly supports asynchrony, because in distributed environments asynchrony typically arises externally from network communication or scheduling.

Definition 2 (Composition). *For modules $M_1 = (X_1, I_1, Q_1, T_1, \lambda_1, q_{0_1})$ and $M_2 = (X_2, I_2, Q_2, T_2, \lambda_2, q_{0_2})$, the composition of M_1 with M_2 is a composite module $M_1|M_2 = (X_1 \cup X_2, (I_1 \cup I_2)\backslash(X_1 \cup X_2), Q, T, \lambda, q_0)$, where*

- *$Q \subseteq Q_1 \times Q_2$ is the maximal set such that $\lambda_1(q_1) \sim \lambda_2(q_2)$ for each state $(q_1, q_2) \in Q$;*
- *$\lambda : Q \rightarrow D^{X_1 \cup X_2}$ labels each state $(q_1, q_2) \in Q$ with the valuation $\lambda_1(q_1) \cup \lambda_2(q_2)$;*
- *T is the minimal transition relation derived by the following composition rules:*

$$\text{ASYN}_L\ \frac{q_1 \xrightarrow{\alpha_1}_{T_1} q_1' \quad q_2 \xrightarrow{\alpha_2}_{T_2} q_2'}{(q_1, q_2) \xrightarrow{\alpha}_T (q_1', q_2)} \qquad \text{ASYN}_R\ \frac{q_1 \xrightarrow{\alpha_1}_{T_1} q_1' \quad q_2 \xrightarrow{\alpha_2}_{T_2} q_2'}{(q_1, q_2) \xrightarrow{\alpha}_T (q_1, q_2')}$$

$$\text{SYN}\ \frac{q_1 \xrightarrow{\alpha_1}_{T_1} q_1' \quad q_2 \xrightarrow{\alpha_2}_{T_2} q_2'}{(q_1, q_2) \xrightarrow{\alpha}_T (q_1', q_2')}$$

  where $\lambda(q_1) \sim \lambda(q_2)$, $\lambda(q_1') \sim \lambda(q_2)$, $\lambda(q_2') \sim \lambda(q_1)$, $\lambda(q_1') \sim \lambda(q_2')$, $\lambda(q_2) \sim \alpha_1$, $\lambda(q_1) \sim \alpha_2$, $\alpha_1 \sim \alpha_2$, and $\alpha = (\alpha_1 \cup \alpha_2)\upharpoonright_I$.
- *$q_0 = (q_{0_1}, q_{0_2}) \in Q$.*

In rule ASYN_L (respectively, ASYN_R) only M_1 (respectively, M_2) evolves; while in rule SYN both M_1 and M_2 evolve simultaneously. In the presence of several modules, the composition rules above can allow only one, some, or all modules evolve simultaneously. The notion of module and composition can be implemented by existing reactive module languages [3,6].

For an infinite word $w = \rho_0\rho_1\ldots$ derived by $M_1|M_2$, we define the notion of stuttering projection to hide asynchronous transitions that do not affect the variables in X_1 or X_2. For $Z \subseteq X_1 \cup X_2$, a *stuttering projection* of w on Z, denoted $w|_Z$, is an infinite word $\rho_0'\rho_1'\ldots$, where there exists $0 = j_0 < j_1 < \cdots$ such that $\rho_i' = \rho_{j_i}\upharpoonright_Z = \rho_{j_i+1}\upharpoonright_Z = \cdots = \rho_{j_{i+1}-1}\upharpoonright_Z$ for any $i \geq 0$. Specifically, the *restriction* of w on Z, denoted $w\upharpoonright_Z$, is the infinite word $\rho_0'\rho_1'\ldots$, where $\rho_i' = \rho_i\upharpoonright_Z$ for any $i \geq 0$.

Thus, a *closed* concurrent system with a finite set X of state variables can be represented as the composition of n modules $M_i = (X_i, I_i, Q_{M_i}, T_{M_i}, \lambda_{M_i}, q_{0_{M_i}})$,

where $X_i \cap X_j = \emptyset$ for any $1 \leq i \neq j \leq n$, $\bigcup_{i=1}^{n} X_i = X$ and $\bigcup_{i=1}^{n} I_i \subseteq X$. D^X is then referred to as the *global* alphabet of the system $M_1|\cdots|M_n$.

Assumptions can then be defined as extended modules with accepting states. In this paper we focus on liveness properties; therefore, we adopt the formalism of Büchi automaton for the definition of assumptions. However, the assume-guarantee rules presented later also apply to safety properties (for which assumptions are then defined as finite automata [4]). We do not discuss safety properties further.

Definition 3 (Assumption). *An assumption is a tuple $A = (X, I, Q, T, \lambda, q_0, F)$, where X, I, Q, T, λ, q_0 are as in Definition 1, and $F \subseteq Q$ is a finite set of accepting states.*

The terminology defined for modules also applies to assumptions. So, an infinite word $\rho_0\rho_1\ldots$ on alphabet D^X is *accepted* by A if there exists an infinite trace $\sigma = q_0\alpha_0q_1\alpha_1\ldots$, referred to as an *accepting trace*, such that $\mathit{inf}(\sigma) \cap F \neq \emptyset$ and $\rho_i = \lambda(q_i)$ for any $i \geq 0$. The *language* $\mathcal{L}(A)$ accepted by A consists of all the infinite words accepted by A. Let coA be the complement of assumption A accepting the complement language $\Omega_X \backslash \mathcal{L}(A)$, where Ω_X is the set of infinite words on alphabet D^X.

The notion of composition can be extended to assumptions. For module $M = (X_1, I_1, Q_1, T_1, \lambda_1, q_{0_1})$ and assumption $A = (X_2, I_2, Q_2, T_2, \lambda_2, q_{0_2}, F_A)$, the composition of M with A is an extended module $M|A = (X, I, Q, T, \lambda, q_0, F)$, where X, I, Q, T, λ, q_0 are as in Definition 2 and $F = \{(q_1, q_2) \in Q \mid q_2 \in F_A\}$. For extended modules $coA_i = (X_i, I_i, Q_i, T_i, \lambda_i, q_{0_i}, F_i)$ $(i = 1, 2)$, the composition of coA_1 with coA_2 is an extended module $coA_1|coA_2 = (X, I, Q, T, \lambda, q_0, F)$, where X, I, Q, T, λ, q_0 are as in Definition 2 and $F = \{(q_1, q_2) \in Q \mid q_1 \in F_1, q_2 \in F_2\}$.

The following definition formalises the notion of guarantee in the context above.

Definition 4 (Guarantee). *For k modules $M_i = (X_i, I_i, Q_i, T_i, \lambda_i, q_{0_i}), 1 \leq k \leq n$, and an assumption $A = (X_A, I_A, Q_A, T_A, \lambda_A, q_{0_A}, F_A)$ such that*

- *$X_i \cap X_j = \emptyset$ for any $1 \leq i \neq j \leq k$;*
- *$M_{i_1}, \ldots, M_{i_{k'}}$ $(1 \leq i_1, \ldots, i_{k'} \leq k)$ are all the $k' \leq k$ modules such that $X_A \cap X_{M_{i_j}} \neq \emptyset$ for $1 \leq j \leq k'$;*
- *$X_A \subseteq \bigcup_{j=1}^{k'} X_{M_{i_j}}$,*

then $M_1|\cdots|M_k$ guarantees *A, denoted $M_1|\cdots|M_k \models A$, if for any infinite word w derived by $M_1|\cdots|M_k$ and any stuttering projection w' of w on $\bigcup_{j=1}^{k'} X_{M_{i_j}}$ that can be derived by $M_{i_1}|\cdots|M_{i_{k'}}$, $w' \lceil_{X_A}$ is accepted by A.*

Note that, if $k' = k$, i.e., $X_A \cap X_i \neq \emptyset$ for any $1 \leq i \leq k$, $M_1|\cdots|M_k \models A$ simply means that for any infinite word w derived by $M_1|\cdots|M_k$, we have that $w \lceil_{X_A}$ is accepted by A.

Simple Assume-Guarantee Rule. Consider the system $M_1|\cdots|M_n$ and a global specification ψ on X that can be represented as the conjunction of local

specifications φ_i on $X_i \cup I_i$ such that $\psi \Leftrightarrow \bigwedge_{i=1}^{n} \varphi_i$. The general assume-guarantee approach [11,4,7,8] either generates assumptions for each module from the global specification and then checks whether these assumptions may collectively violate it (as shown by rule SYM below); or generates an assumption for some module (e.g., M_1) from the global specification and then checks whether the assumption can be guaranteed by the rest of modules (as shown by rule ASYM below).

$$\text{SYM}\ \frac{\forall 1 \le i \le n,\ M_i|A_i \ \vDash \psi \qquad \mathcal{L}(coA_1|\cdots|coA_n) = \emptyset}{M_1|\cdots|M_n \ \vDash \psi} \qquad \text{ASYM}\ \frac{M_1|A_1 \ \vDash \psi \qquad M_2|\cdots|M_n \vDash A_1}{M_1|\cdots|M_n \ \vDash \psi}$$

Thus, it is a common practice to generate assumptions from global specifications. However, in concurrent systems, each module typically control its state variables under inputs from *only a small* proportion of other modules. Therefore, in standard methodologies:

- each assumption A_i for module M_i may contain irrelevant valuations of state variables that module M_i does not depend on. This makes the size of assumption A_i larger than necessary.
- whenever the system is extended, each assumption A_i may have to be modified to incorporate the state variables of the additional modules. Hence, assumptions already generated for the existing modules cannot be reused for verifying the extended system.

We propose to avoid these issues by assigning each module M_i with the corresponding local specification φ_i. Inspired by rules SYM and ASYM, we present below rules $\mathbf{R_0}$ and $\mathbf{R_1}$, respectively. Recall that $\psi \Leftrightarrow \bigwedge_{i=1}^{n} \varphi_i$.

$$\mathbf{R_0}\ \frac{\forall 1 \le i \le n,\ M_i|A_i \ \vDash \varphi_i \qquad \mathcal{L}(coA_1|\cdots|coA_n) = \emptyset}{M_1|\cdots|M_n \ \vDash \bigwedge_{i=1}^{n} \varphi_i} \qquad \mathbf{R_1}\ \frac{\forall 1 \le i \le n,\ M_i|A_i \ \vDash \varphi_i \qquad M_{i_1}|\cdots|M_{i_{k_i}} \vDash A_i}{M_1|\cdots|M_n \ \vDash \bigwedge_{i=1}^{n} \varphi_i}$$

In rules SYM and ASYM assumptions A_i are all supposed to be generated from the global specification ψ (on X); while in rules $\mathbf{R_0}$ and $\mathbf{R_1}$ each assumption A_i is to be generated from the corresponding local specification φ_i (on $X_i \cup I_i$). In this way the size of assumption A_i can be reduced because only variables in $X_i \cup I_i$ (which is a subset of X) have to be concerned with assumption A_i.

Unsound Rule $\mathbf{R_0}$. As a side effect in rule $\mathbf{R_0}$, assumption A_i may admit more interactions with module M_i than can be admitted by the assumptions generated from the global specification ψ. This is because the variables in $X \backslash (X_i \cup I_i)$ are not constrained by the local specification φ_i. Therefore, the tentative rule $\mathbf{R_0}$ above does not preserve soundness, though its completeness is not affected by the weaker assumptions. We refer to our technical report [18] for a counterexample where rule $\mathbf{R_0}$ fails.

Sound Rule $\mathbf{R_1}$. Modules $M_{i_1}, \ldots, M_{i_{k_i}}$ ($k_i \ge 1$) in rule $\mathbf{R_1}$ are all the k_i neighbours of module M_i that control its input variables in I_i (i.e., $I_i \subseteq \bigcup_{j=1}^{k_i} X_{i_j}$

and $I_i \cap X_{i_j} \neq \emptyset$ for any $1 \leq j \leq k_i$). Theorem 1 shows the soundness of rule $\mathbf{R_1}$ for local specifications.

Theorem 1 (Soundness). *If for any module M_i $(1 \leq i \leq n)$ there exists an assumption A_i such that $M_i|A_i \vDash \varphi_i$ and $M_{i_1}|\cdots|M_{i_{k_i}} \vDash A_i$, then $M_1|\cdots|M_n \vDash \bigwedge_{i=1}^{n} \varphi_i$.*

Proof. By contradiction. Consider an infinite word $w = \rho_0\rho_1\ldots$ on the global alphabet (i.e., each ρ_i is a valuation on X) that makes the conclusion fail on some φ_j $(1 \leq j \leq n)$. Then, since the state variables in X_j are exclusively controlled by M_j, any stuttering projection $w|_{X_j \cup I_j}$ would not be accepted by $M_j|A_j$ and hence any stuttering projection $w|_{I_j}$ would not be accepted by A_j.

However, the variables in X_{j_l} $(1 \leq l \leq k_j)$ are exclusively controlled by M_{j_l}. By the composition rules in Definition 2, there exists a stuttering projection of w on $\bigcup_{l=1}^{k_j} X_{j_l}$, denoted w', that is derived by $M_{j_1}|\cdots|M_{j_{k_j}}$. Since $I_j \subseteq \bigcup_{l=1}^{k_j} X_{j_l}$ and $M_{j_1}|\cdots|M_{j_{k_j}} \vDash A_j$, we have that $w'\lceil_{I_j}$ is accepted by A_j. This is a contradiction because $w'\lceil_{I_j}$ is also a stuttering projection of w on I_j.

Unfortunately, rule $\mathbf{R_1}$ is not complete. In fact, for each module M_i, its neighbour modules are isolated from the system when being examined against assumption A_i. This ignores the impact of the rest of modules on its neighbour modules. For example, consider a system consisting of the following four modules M_i $(1 \leq i \leq 4)$:

M_i	X_i	I_i	Transition Function
M_1	$\{x_1\}$	$\{x_2, x_3\}$	$x_1' = x_2 - x_3$
M_2	$\{x_2\}$	$\{x_4\}$	$x_2' = x_2 - x_4$
M_3	$\{x_3\}$	$\{x_4\}$	$x_3' = x_3 + x_4$
M_4	$\{x_4\}$	$\{x_2, x_3\}$	$x_4' = \begin{cases} 1 & x_2 > x_3 \text{ and } x_4 > 0 \\ -1 & x_2 < x_3 \text{ and } x_4 < 0 \\ 0 & \text{otherwise} \end{cases}$

Let x' be the next value of variable x. Then, for each module M_i, the CTL formula $AFAG\ (\bigwedge_{x \in X_i \cup I_i} (x' = x))$ specifies that the values of the variables in $X_i \cup I_i$ will always eventually remain unchanged for ever.

With an initial state $(x_1, x_2, x_3, x_4) = (u-v, u, v, 1)$ for any $u > v \geq 0$, it can be seen that $M_1|M_2|M_3|M_4 \vDash \bigwedge_{i=1}^{4} AFAG\ (\bigwedge_{x \in X_i \cup I_i} (x' = x))$. This is because x_2 and x_3 evolve by converging in a step of size x_4, until x_2 and x_3 meet or just cross over each other. Then, the system $M_1|M_2|M_3|M_4$ reaches a stable state where $x_4 = 0$.

However, by $M_2|M_3$ itself, x_2 and x_3 may diverge from each other. Hence, such divergent sequence of inputs (x_2, x_3) cannot lead M_1 to stabilising x_1, and so cannot be accepted by any assumption A_1 that satisfies the premise $M_1|A_1 \vDash AFAG\ \bigwedge_{x \in X_1 \cup I_1} (x' = x)$.

3 Bounded Assume-Guarantee Reasoning

In this section we modify rule $\mathbf{R_1}$ to achieve completeness by exploiting the neighbourhood dependency between modules. This results in a "bounded" rule $\mathbf{R}^{\pi}$, which defines a bounded assume-guarantee reasoning approach.

Let $\mathcal{D} = \{(M_1, M_2) \mid X_2 \cap I_1 \neq \emptyset\}$ be the direct dependency relation between the modules of the system $M_1|\cdots|M_n$. $(M_1, M_2) \in \mathcal{D}$ means that module M_1 depends on the inputs from (or reacts directly with) module M_2. Then, the k-dependency relation $\mathcal{D}^k$ is defined recursively as follows: $\mathcal{D}^1 = \mathcal{D}$ and $\mathcal{D}^k = \mathcal{D}^{k-1} \cup (\mathcal{D}^{k-1} \circ \mathcal{D})$ for $k > 1$, where $\mathcal{D}^{k-1} \circ \mathcal{D}$ is the composition of $\mathcal{D}^{k-1}$ with $\mathcal{D}$.

For module M_i let $\mathcal{N}_i^k$ be the set of all the modules M except M_i such that $(M_i, M) \in \mathcal{D}^k$, and $\mathcal{C}_i^k$ be the composition of all the modules in $\mathcal{N}_i^k$. Then, rule $\mathbf{R_1}$ can be extended further to rule $\mathbf{R_k}$ as follows:

$$\mathbf{R_k}\,\frac{\forall 1 \le i \le n, \begin{array}{c} M_i|A_i \models \varphi_i \\ \mathcal{C}_i^k \models A_i \end{array}}{M_1|\cdots|M_n \models \bigwedge_{i=1}^{n} \varphi_i}$$

Informally, for each module M_i, rule $\mathbf{R_1}$ involves reasoning about its neighbour modules only; while rule $\mathbf{R_k}$ checks *all* the modules within the range of k hops around module M_i. Similarly, it can be proved that rule $\mathbf{R_k}$ is sound for any $k \ge 1$.

Theorem 2 (Soundness). *Given $k \ge 1$, if for any module M_i ($1 \le i \le n$), there exists an assumption A_i such that $M_i|A_i \models \varphi_i$ and $\mathcal{C}_i^k \models A_i$, then $M_1|\cdots|M_n \models \bigwedge_{i=1}^{n} \varphi_i$.*

Proof. By contradiction. The proof is similar to that of Theorem 1.

If the modules within k hops around module M_i can together guarantee assumption A_i, then such guarantee is preserved by the modules within $k+1$ hops. This is because assumption A_i can already be guaranteed regardless of the interactions with the additional modules. Based on this observation, Theorem 3 relates rule $\mathbf{R_k}$ with rule $\mathbf{R_{k+1}}$.

Theorem 3. *Let A_i be an assumption for module M_i. Then, $\mathcal{C}_i^k \models A_i$ implies $\mathcal{C}_i^{k+1} \models A_i$.*

Proof. By the definition of $\mathcal{D}^k$, we have $\mathcal{N}_i^k \subseteq \mathcal{N}_i^{k+1}$. So, $I_i \subseteq \bigcup_{M_j \in \mathcal{N}_i^k} X_j \subseteq \bigcup_{M_j \in \mathcal{N}_i^{k+1}} X_j$. For any infinite word w derived by $\mathcal{C}_i^{k+1}$, there exists a stuttering projection of w on $\bigcup_{M_j \in \mathcal{N}_i^k} X_j$, denoted w', that can be derived by $\mathcal{C}_i^k$. Since $\mathcal{C}_i^k \models A_i$, $w' \lceil_{I_i}$ would be accepted by A_i for any such w'.

Since the system consists of a finite number of state variables, there exists a transitive dependency closure $\mathcal{D}^{\pi}$ ($\pi \ge 1$) such that $\mathcal{D}^{\pi} = \mathcal{D}^{\pi+1}$. Theorem 4 shows that rule $\mathbf{R_{\pi}}$ is complete for local specifications.

Theorem 4 (Completeness). *Suppose $\mathcal{D}^\pi$ is the transition dependency closure of the system $M_1|\cdots|M_n$. If $M_1|\cdots|M_n \models \bigwedge_{i=1}^{n} \varphi_i$, then for each module M_i, there exists an assumption A_i such that $M_i|A_i \models \varphi_i$ and $\mathcal{C}_i^\pi \models A_i$.*

Proof. By construction. For each module M_i, $\mathcal{C}_i^\pi$ could be extended as such assumption A_i by appointing all states in $\mathcal{C}_i^\pi$ as accepting states. This is because for any $1 \leq j \leq k$, $\bigwedge_{i=1}^{n} \varphi_i$ implies φ_j, and the variables in X_j are exclusively controlled by M_j that is independent of modules not in $M_j|\mathcal{C}_j^\pi$.

As a corollary of theorems 2, 3 and 4, rule R_π could be reformulated as rule R^π below, which is also sound and complete for local specifications.

$$\mathbf{R}^\pi \frac{\forall 1 \leq i \leq n, \begin{array}{r} M_i|A_i \models \varphi_i \\ \exists 1 \leq d_i \leq \pi,\ \mathcal{C}_i^{d_i} \models A_i \end{array}}{M_1|\cdots|M_n \models \bigwedge_{i=1}^{n} \varphi_i}$$

Rule $\mathbf{R}^\pi$ can be applied incrementally for compositional verification of concurrent systems. For the sake of generality and reusability, we opt for the weakest assumption WA_i [4,22] that admits as many as possible sequences of inputs to module M_i without violating the local specification φ_i. For module M_i, the weakest assumption WA_i is an assumption such that

- $\mathcal{L}(WA_i) \subseteq \mathcal{I}(M_i)$ and $M_i|WA_i \models \varphi_i$;
- $\mathcal{L}(A_i) \subseteq \mathcal{L}(WA_i)$ for any assumption A_i such that $\mathcal{L}(A_i) \subseteq \mathcal{I}(M_i)$ and $M_i|A_i \models \varphi_i$.

Then, the verification task for checking whether the system $M_1|\cdots|M_n$ satisfies the global specification ψ ($\Leftrightarrow \bigwedge_{i=1}^{n} \varphi_i$) can be decomposed into n parallel sub-tasks. For each pair of module M_i and local specification φ_i, we envisage the following procedure:

1: Generate the weakest assumption WA_i from the local specification φ_i;
2: $d_i \leftarrow 1$;
3: **while** $\mathcal{C}_i^{d_i} \not\models WA_i$ **do**
4:     **if** $\mathcal{N}_i^{d_i} \neq \mathcal{N}_i^{d_i+1}$ **then**
5:         $d_i \leftarrow d_i + 1$;
6:     **else**
7:         **return false**;
8: **return true**;

The weakest assumption WA_i is suitable for checking an increasing number of modules as the while-loop continues (Line 3). Since the number of modules is finite, this procedure will terminate: either the assumption WA_i is guaranteed (Line 8), or all the modules that M_i reacts with have been checked (Line 7).

4 Case Study

One of our motivations for investigating assume-guarantee reasoning was to broaden the range of applications in the area of network control. We particularly

wish to reason about the *overall* objectives or behaviour of the control algorithm implemented by a protocol. This section illustrates an application of rule $\mathbf{R}^{\pi}$ to verify the stability of an optimisation-based congestion control system. Both the dynamic system and the stability property exhibit compositional structures. We refer to our previous work [17] for more details about the system and the property we considered.

Multi-Path Congestion Control. For tractability, we devise a discrete version of the fluid-flow congestion control algorithm proposed by Kelly and Voice [15].

Consider a network in which a finite number of sources communicate with a finite number of destinations. Between each pair of source and destination a number of routes have been provisioned. Let $r \in s$ denote that route r is available to source s and $s(r)$ be the source that transmits along route r. Each route uses a number of links or, more generally, resources, each of which has a finite capacity constraint. Let $j \in r$ denote that resource j is used by route r.

Then, for each source s and route r available to s, the discrete trajectory in the flow rate x_r is subject to the following equation:

$$x'_r = x_r + \kappa_r x_r \left(1 - \frac{1}{\alpha_{s(r)}} \sum_{j \in r} \beta_j x_j \sum_{r' \in s(r)} x_{r'}\right)^+_{x_r} \tag{1}$$

where κ_r is a constant, x'_r is the next value of x_r and

- α_s is the utility co-efficient of source s;
- β_j is the price co-efficient of resource j;
- x_j is the aggregate flow rate at resource j, i.e., $x_j = \sum_{j \in r} x_r$;
- $(z)^+_x = \min(0, z)$ if $x \leq 0$, otherwise $(z)^+_x = z$.

Thus, each source s adjusts the flow rate x_r on route $r \in s$ based on feedback $\beta_j x_j$ from every resource $j \in r$ in the network (indicating congestion). Then, the algorithm presented in [15] is composed of these sources acting synchronously and collectively. The stability of this synchronous algorithm has been proved in [15]. Herein, we analyse the fully asynchronous variant of the algorithm under the fairness constraint that every source acts infinitely often. This asynchronous model captures uncertain delay between distributed sources.

Stability. System stability is a key property of interest for a distributed congestion control system. A system is stable if it equilibrates at certain network-wide flow configuration, i.e., where $x'_r = x_r$ for every route r. Let s_i range over all the sources. Then, the following CTL formula

$$AFAG \bigwedge_{s_i} \left(\bigwedge_{r \in s_i} x'_r = x_r \right) \tag{2}$$

represents system stability, that is, the system will always eventually be stable.

Lagrangian decomposition techniques reduce system stability onto individual modules [19]. A distributed source is stable if certain stable flow configuration

is reached on all the routes using the resources consumed by the source. Let $\gamma(s_i)$ denote the set of the routes serving or sharing resource with source s_i, i.e., $\gamma(s_i) = \{r \mid j \in r \text{ for any } r' \in s_i \text{ and } j \in r'\}$. Then, local stability on source s_i is represented by the following CTL formula

$$AFAG\ (\bigwedge_{r\in\gamma(s_i)} x'_r = x_r) \tag{3}$$

Observe that the global specification (2) is equivalent to the conjunction of local specifications (3) on all the sources. Therefore, we instantiate rule $\mathbf{R}^\pi$ as rule **SS** below for **s**ystem **s**tability:

$$\mathbf{SS}\ \frac{\begin{array}{c}\forall 1 \le i \le n, \ M_i|A_i \ \models AFAG \bigwedge_{r\in\gamma(s_i)} x'_r = x_r \\ \exists 1 \le d_i \le \pi,\ \mathcal{C}_i^{d_i} \models A_i\end{array}}{M_1|\cdots|M_n \ \models \ AFAG \bigwedge_{s_i}(\bigwedge_{r\in s_i} x'_r = x_r)}$$

where source s_i is represented as module M_i.

Computing Assumptions. By rule **SS**, the assumption A_i for module M_i is such that $M_i|A_i$ satisfies the local specification (3). Thus, assumption A_i concerns only on the variables in $X_i \cup I_i$, and is meant to supply sequences of inputs to module M_i such that $M_i|A_i$ can eventually converge to certain configuration on $X_i \cup I_i$.

Note that under rule SYM or ASYM, assumption A_i has to concern on all the variables in X. A local stable state on $X_i \cup I_i$ would be extended to a global stable states on X to meet the global specification (2). Since module M_i controls only the variables in X_i, all the variables in $X \backslash (X_i \cup I_i)$ can converge to any possible combinations of values in domain D. Hence, for every local stable state on $X_i \cup I_i$, assumption A_i has to cover all the corresponding $|D|^{|X\backslash(X_i\cup I_i)|}$ global stable states. Such redundancy is avoided under rule **SS** by generating assumption A_i from the local specification (3).

For module $M_i = (X_i, I_i, Q_{M_i}, T_{M_i}, \lambda_{M_i}, q_{0_{M_i}})$, the assumption can be constructed as a tuple $A_i = (I_i, X_i, E_{A_i} \cup F_{A_i}, T_{A_i}, \lambda_{A_i}, q_{0_{A_i}}, F_{A_i})$ where E_{A_i}, F_{A_i}, T_{A_i} and λ_{A_i} are the minimal sets of non-accepting states, accepting states, transitions and the labelling function derived through the following algorithm, respectively.

1. For each valuation α on I_i, there exists one and only one state $p \in E_{A_i}$ such that $\lambda_{A_i}(p) = \alpha$.
2. For any $q \xrightarrow{\alpha}_{M_i} q'$ and the state $p \in E_{A_i}$ such that $\lambda_{A_i}(p) = \alpha$, $p \xrightarrow{\lambda_{M_i}(q)}_{A_i} p'$ for all $p' \in E_{A_i}$.
3. For any $q \xrightarrow{\alpha}_{M_i} q$, there exists one and only one state $p_q \in F_{A_i} \backslash E_{A_i}$ such that $\lambda_{A_i}(p_q) = \alpha$ and
   - $p_q \xrightarrow{\lambda_{M_i}(q)}_{A_i} p_q$;
   - $p \xrightarrow{\lambda_{M_i}(q)}_{A_i} p_q$, where $p \in E_{A_i}$ is the state such that $\lambda_{A_i}(p) = \alpha$;
4. $q_{A_{i_0}}$ is the initial state, and $\lambda(q_{A_{i_0}})$ is the given initial configuration on I_i.

Intuitively, step 1 logs all possible inputs to module M_i as the non-accepting states of assumption A_i; while step 2 traces the state changes of module M_i as the transitions of assumption A_i. Step 3 defines the accepting states of assumption A_i to characterise all configuration on $X_i \cup I_i$ where $M_i|A_i$ can possibly settle. Each self-loop transition $q \xrightarrow{\alpha}_{M_i} q$ contributes to an accepting state p_q with $\lambda_{A_i}(p_q) = \alpha$. Apparently, module M_i at state q would remain at this state under constantly repeated inputs α, which is exactly what the local specification (3) expects.

Thus, we compute an assumption A_i for module M_i based on the module itself, regardless of the underlying topology. Theorem 5 shows that the assumption is an appropriate one for our purpose.

Theorem 5. *Assumption A_i generated by the above algorithm for module M_i is the weakest assumption with respect to the local specification* (3)*.*

Proof. By definition, it can be seen that any accepting trace of $M_i|A_i$ will fall into an infinite loop at some state (q, p_q), where $q \in Q_{M_i}$ admits a self-loop transition under input $\lambda_{A_i}(p_q)$. Correspondingly, the infinite word accepted through such an accepting trace will terminate with an infinite loop of the valuation on $\lambda_{M_i}(q) \cup \lambda_{A_i}(p_q)$. Therefore, $M_i|A_i$ satisfies the local specification (3).

We then prove by contradiction that assumption A_i is the weakest assumption with respect to the local specification (3). Suppose there exists an assumption A'_i such that $\mathcal{L}(A'_i) \subseteq \mathcal{I}(M_i)$ and $M_i|A'_i$ satisfies the local specification (3), but there exists an infinite word $\theta = \alpha_0\alpha_1 \cdots \in \mathcal{L}(A'_i)$ that is not accepted by A_i. Then, by this hypothesis and the definition of step 3, θ cannot be derived by A_i.

Assume $\alpha_0 \ldots \alpha_k$ $(k \geq 0)$ is the longest prefix that can be derived from A_i. This means that, for any valuation ρ on X_i, no transition $p \xrightarrow{\rho}_{A_i} p'$ exists such that $\lambda_{A_i}(p) = \alpha_k$ and $\lambda_{A_i}(p') = \alpha_{k+1}$. Hence, by the definition of step 2, no transition $q \xrightarrow{\alpha_k}_{M_i} q'$ exists such that for any states $q, q' \in Q_{M_i}$. This conflicts with the hypothesis, which implies $\theta \in \mathcal{I}(M_i)$.

The time complexity of this algorithm is linear to the size of module M_i. The worst run-time is $O(2|T_{M_i}|)$. The size of the resulting assumption A_i is also linear to the size of module M_i. In the worst case, assumption A_i contains $|D|^{|I_i|} + |T_{M_i}|$ number of states and $|T_{M_i}||D|^{|I_i|} + 2|T_{M_i}|$ number of transitions.

By omitting step 4, this algorithm can be revised to generate a *super* assumption with the universal set of all possible initial states, each labelled with a valuation on I_i. The language accepted by the super assumption is then the disjoint union of the languages accepted by the assumptions under each possible initial valuation on I_i.

5 Experiments

This section illustrates how reduced assumptions can help improve the efficiency and scalability of assume-guarantee reasoning. Specifically, we show how one set of verification checks under rule **SS** can prove stability regardless of the number

of sources and their initial flow configurations, and for any topology of bounded degree.

Consider a simple topology where each source is provisioned with two routes and each resource is shared by two sources. Thus, each source module has two state variables and two input variables. Let $M_{u,v}$ be a source with an initial configuration $(u, v) \in D^2$ and the transitions defined by Equation (1). Then, no matter how many sources a network may consist of, each source is of the general form $M_{u,v}$, where $u, v \in D$.

Let $A_{u,v}$ be the super assumption generated by the above algorithm for module $M_{u,v}$. We start with checking whether the composition of any two possible neighbour modules can guarantee these assumptions. This amounts to check whether

$$M_{u_1,v_1'} | M_{u_1',v_1} \models A_{u_0,v_0} \tag{4}$$

for any initial configuration $(u_0, v_0, u_1, v_1, u_1', v_1') \in D^6$. For the domain $D = [1, 6]$ this means that $6^6 (= 46656)$ instances of Equation (4) need to be verified. These checks are done through, as usual, by establishing whether any infinite word derived by $M_{u_1,v_1'} | M_{u_1',v_1}$ can be accepted by coA_{u_0,v_0}, the complement of assumption A_{u_0,v_0}.

We use the GOAL tool [25] to compute and simplify each complement coA_{u_0,v_0}. Each assumption A_{u_0,v_0} and its complement coA_{u_0,v_0} are encoded as Büchi automata in GOAL. Table 1 reports the size of each automaton in terms of the number of states (in Columns *#states*) and the number of transitions (in Columns *#transitions*), and the time usage in seconds for complementing each assumption A_{u_0,v_0} (in Column *time*). Note that M_{v_0,u_0} is equivalent to M_{u_0,v_0} under permutation. For sake of comparison, Table 1 also reports the size of each assumption $A^{\psi}_{u_0,v_0}$, generated from the global specification (2), and the time usage (in seconds) for complementing it. The symbol '-' means that the tool did not return any result within 10 hours. All experiments were ran on a Linux server with two Intel 2.8GHz Quad Core Xeon processors and 16G memory. Observe that GOAL is not a tool optimised for speed; faster results may possibly be achievable.

It can be seen that assumptions for each module M_{u_0,v_0} are greatly reduced under rule **SS**. In average each assumption A_{u_0,v_0} is reduced by a factor of 36 times in the number of states and a factor of 569 in the number of transitions compared with the corresponding assumption $A^{\psi}_{u_0,v_0}$. This is because the combinatorial explosion with the redundant variables in $X \backslash (X_i \cup I_i)$ for each module M_i is avoided without loss of expressiveness of the assumptions. The advantage of using reduced assumptions is particularly apparent when computing their complements. The tool took no more than 2.5 minutes to complement each assumption A_{u_0,v_0}, but only 10 out of 21 complementation instances $coA^{\psi}_{u_0,v_0}$ can be computed by the tool. Considering that it is very time-consuming to simplify a Büchi automaton, our approach is more efficient than that of applying the general assume-guarantee rules with simplified assumptions $A^{\psi}_{u_0,v_0}$.

Equation (4) was verified in our experiments for all the values of parameters $u_0, v_0, u_1, v_1, u_1', v_1'$ in domain D. As a consequence, any assumption A_{u_0,v_0} can

Table 1. Experimental Results for Computing Assumptions

u_0	v_0	$A^{\psi}_{u_0,v_0}$			A_{u_0,v_0}			coA_{u_0,v_0}	
		#states	#transitions	time(s)	#states	#transitions	time(s)	#states	#transitions
1	1	1332	49248	1511.0	37	108	3.3	73	2628
1	2	1332	49248	1475.1	37	108	1.9	73	2628
1	3	1332	49248	1415.8	37	108	1.7	73	2628
1	4	2016	97200	3292.9	56	180	3.5	110	3960
1	5	2268	144288	4247.5	63	228	4.9	123	4428
1	6	2304	190944	5693.8	64	264	5.0	124	4464
2	2	1332	49248	1477.2	37	108	1.6	73	2628
2	3	4752	195840	14207.2	132	400	19.3	114	4104
2	4	5760	291024	21088.4	160	524	31.5	168	6048
2	5	5796	337680	25180.2	161	560	33.1	169	6084
2	6	6084	431424	-	169	644	34.6	183	6588
3	3	8532	389736	-	237	746	77.3	174	6264
3	4	9648	531720	-	268	910	103.5	233	8388
3	5	9684	578376	-	269	946	106.4	234	8424
3	6	9756	671688	-	271	1018	105.9	236	8496
4	4	8568	436392	-	238	782	74.7	175	6300
4	5	9684	578376	-	269	946	108.1	234	8424
4	6	9684	578376	-	269	946	104.1	234	8424
5	5	10836	767016	-	301	1146	145.1	294	10584
5	6	10836	767016	-	301	1146	138.0	294	10584
6	6	10836	767016	-	301	1146	138.1	294	10584

be guaranteed by the composition of any two possible modules. Thus, our experiments show the stability of such system for *any* number of sources and *any* initial flow configuration under the given topology.

Furthermore, the experiments reported can be extended for any topology with bounded degree (i.e., each source is sharing resources with a bounded number of other sources). Suppose each source has at most m routes, the general form of each module is $M_{\boldsymbol{u}}$, where vector $\boldsymbol{u}$ ranges over $\bigcup_{k=1}^{m} D^k$. This is particularly appealing to us as previous results in the literature on verification of congestion control models (e.g., [26,17]) apply only to fixed network topologies.

6 Conclusions

The paper presents a methodology for assume-guarantee reasoning for global specifications consisting of conjunctions of local specifications. The rule $\mathbf{R}^{\pi}$ presented is both sound and complete for local specifications, yet can be applied to draw conclusions on global specifications. Thus, a verification task on a system can be decomposed onto individual modules and local specifications. The methodology is based on an incremental approach to exploit the neighbourhood dependency between modules. Each increment explores the modules' interactions one step further into the neighbourhood.

We applied the rule to verify the stability of a distributed congestion control system with any number of modules, any initial state, and any topology of

bound degree. We proved system stability by considering only local stability of each individual source when interacting with its neighbours. In this way, the technique presented could greatly extend the range of network problems that model checking could be applied to.

References

1. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems 17(3), 507–535 (1995)
2. Alur, R., Henzinger, T.A.: Reactive modules. In: Proc. 11th Annual IEEE Symposium on Logic in Computer Science Logic in Computer Science (LICS 1996), New Brunswick, USA, July 27-30, pp. 207–218 (1996)
3. Alur, R., Henzinger, T.A., Mang, F., Qadeer, S., Rajamani, S.K., Tasiran, S.: MOCHA: Modularity in model checking. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
4. Barringer, H., Giannakopoulou, D., Păsăreanu, C.S.: Proof rules for automated compositional verification through learning. In: Proc. 2003 Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2003), Helsinki, Finland, September 1-2, pp. 14–21 (2003)
5. Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional reasoning in model checking. In: Revisted Lectures from Proc. International Symposium on Compositionality, Bad Malente, Germany, September 8-12, pp. 81–102 (1997)
6. 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, Springer, Heidelberg (2002)
7. Cobleigh, J., Giannakopoulou, D., Păsăreanu, C.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
8. Farzan, A., Chen, Y.F., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
9. Francez, N., Pnueli, A.: A proof method for cyclic programs. Acta Informatica 9(2), 133–157 (1978)
10. Gheorghiu Bobaru, M., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
11. Giannakopoulou, D., Păsăreanu, C.S., Barringer, H.: Assumption generation for software component verification. In: Proc. 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, UK, September 23-27, pp. 3–12 (2002)
12. Grumberg, O., Long, D.E.: Model checking and modular verification. ACM Transactions on Programming Languages and Systems 16(3), 843–871 (1994)
13. Henzinger, T.A., Qadeer, S., Rajamani, S.K.: You assume, we guarantee: Methodology and case studies. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 440–451. Springer, Heidelberg (1998)
14. Jones, C.B.: Tentative steps toward a development method for interfering programs. ACM Transactions on Programming Languages and Systems 5(4), 596–619 (1983)

15. Kelly, F., Voice, T.: Stability of end-to-end algorithms for joint routing and rate control. ACM SIGCOMM Computer Communication Review 35(2), 5–12 (2005)
16. Kupferman, O., Vardi, M.Y.: An automata-theoretic approach to modular model checking. ACM Transactions on Programming Languages and Systems 22(1), 87–128 (2000)
17. Lomuscio, A., Strulo, B., Walker, N., Wu, P.: Model checking optimisation-based congestion control models. In: Proc. 2009 Workshop on Concurrency, Specification, and Programming (CS&P 2009), Kraków-Przegorzały, Poland, September 28-30, pp. 386–397 (2009)
18. Lomuscio, A., Strulo, B., Walker, N., Wu, P.: Assume-guarantee verification for distributed systems with local specifications. Tech. Rep. RN/10/01, Department of Computer Science, University College London (February 2010)
19. Low, S.H., Lapsley, D.E.: Optimization flow control, I: basic algorithm and convergence. IEEE/ACM Transactions on Networking 7(6), 861–874 (1999)
20. Maier, P.: Compositional circular assume-guarantee rules cannot be sound and complete. In: Gordon, A.D. (ed.) FOSSACS 2003. LNCS, vol. 2620, pp. 343–357. Springer, Heidelberg (2003)
21. Misra, J., Chandy, K.M.: Proof of networks of processes. IEEE Transactions on Software Engineering SE 7(4), 417–426 (1981)
22. 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)
23. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. Formal Methods in System Design 32(3), 207–234 (2008)
24. Păsăreanu, C.S., Giannakopoulou, D., Bobaru, M.G., Cobleigh, J.M., Barringer, H.: Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. Formal Methods in System Design 32(3), 175–205 (2008)
25. Tsay, Y.K., Chen, Y.F., Tsai, M.H., Wu, K.N., Chan, W.C.: GOAL: A graphical tool for manipulating büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 466–471. Springer, Heidelberg (2007)
26. Yuen, C., Tjioe, W.: Modeling and verifying a price model for congestion control in computer networks using promela/spin. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 272–287. Springer, Heidelberg (2001)

Automating Coinduction with Case Analysis

Eugen-Ioan Goriac[1], Dorel Lucanu[1], and Grigore Roşu[2]

[1] Faculty of Computer Science,
Alexandru Ioan Cuza University, Romania
{egoriac,dlucanu}@info.uaic.ro
[2] Department of Computer Science,
University of Illinois at Urbana-Champaign, USA
grosu@cs.uiuc.edu

Abstract. Coinduction is a major technique employed to prove behavioral properties of systems, such as behavioral equivalence. Its automation is highly desirable, despite the fact that most behavioral problems are Π_2^0-complete. Circular coinduction, which is at the core of the CIRC prover, automates coinduction by systematically deriving new goals and proving existing ones until, hopefully, all goals are proved. Motivated by practical examples, circular coinduction and CIRC have been recently extended with several features, such as special contexts, generalization and simplification. Unfortunately, none of these extensions eliminates the need for case analysis and, consequently, there are still many natural behavioral properties that CIRC cannot prove automatically. This paper presents an extension of circular coinduction with case analysis constructs and reasoning, as well as its implementation in CIRC. To uniformly prove the soundness of this extension, as well as of past and future extensions of circular coinduction and CIRC, this paper also proposes a general correct-extension technique based on equational interpolants.

1 Introduction

Automated theorem proving is a subject of high interest in computer science, frequently used in industry for hardware and software verification. Coinduction is a proof technique for properties over infinite data structures (which typically model behaviors of reactive systems) or for behavioral properties. Since coinduction is too complex to be automated in its full generality, existing tools attempt to implement simpler, algorithmic variants which work in many practical cases. Such a tool is CIRC [7], which implements circular coinduction [4,10]. Circular coinduction has been recently extended with special contexts [8], generalization and simplification rules [6]. Many computer experiments with CIRC led us to the necessity of introducing and automating case reasoning.

Case analysis is a fundamental algebraic/coalgebraic reasoning technique whose importance has been early noticed and which has been partly investigated (see, e.g., [1,5]). Automating case analysis in its full generality is a difficult task, which would certainly lead to expensive, non-terminating procedures. In this paper we investigate practical means to incorporate limited but effective support for automatic case analysis in coinductive provers, such as CIRC.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 220–236, 2010.
© Springer-Verlag Berlin Heidelberg 2010

We start by discussing two motivating examples that emphasize the importance of case analysis when proving properties by coinduction.

A *stream* is an infinite-list data structure $a_1 : a_2 : a_3 \ldots$ which can be used to model infinite behaviors. Considering the stream observers *hd* and *tl*, defined by $hd(a : s) = a$ and $tl(a : s) = s$, to prove a stream equality $s = s'$ by coinduction one needs to find a set of pairs $R = \{u_i \equiv v_i \mid i \in I\}$, which contains the pair $s \equiv s'$ and which is a congruence with respect to *hd* and *tl*, i.e., $u \equiv v \in R$ implies $hd(u) = hd(v)$ and $tl(u) \equiv tl(v) \in R$ [11,10].

Example 1. We present a situation where case analysis over a term of enumerable sort is needed. We assume that the bitwise negation operator, *not*, is defined over streams of bits using the auxiliary bit-complement operation $\overline{\cdot}$. The function f creates an infinite alternating bit stream, starting with a given first element:

$$\begin{array}{ll} \overline{0} = 1 \quad \overline{1} = 0 & hd(f(a)) = a \\ hd(not(s)) = \overline{hd(s)} & hd(tl(f(a))) = \overline{a} \\ tl(not(s)) = not(tl(s)) & tl(tl(f(a))) = f(a) \end{array}$$

Above, s is a variable of sort stream and a is a variable of sort bit. Let us prove $f(\overline{a}) = not(f(a))$ by coinduction. Take $R = \{f(\overline{a}) \equiv not(f(a)), tl(f(\overline{a})) \equiv tl(not(f(a)))\}$. For the first pair, $hd(f(\overline{a})) = hd(not(f(a)))$ holds. This property can only be checked by making a case analysis over a, which takes values from $\{0, 1\}$. Further, $tl(f(\overline{a})) \equiv tl(not(f(a)))$ is reduced to $tl(f(\overline{a})) \equiv not(tl(f(a))) \in R$. For the second pair, the reasoning is similar. When checking for congruence w.r.t. *hd*, another case analysis over a is needed.

We show in this paper that, when the system knows that certain terms are of enumerable sort, case analysis can be done automatically. Case analysis based on enumerated sorts can be seen as a particular case of induction. However, we prefer to treat it separately because its integration with the circular coinduction engine is much simpler and, consequently, more efficient.

Example 2. The second example shows how to prove a property over streams of integers. We define the operator *sign* which, when provided a stream of integers, returns another stream with elements from the set $\{-1, 0, 1\}$:

$$hd(sign(s)) = \begin{cases} -1 & \texttt{if } hd(s) < 0 \\ 0 & \texttt{if } hd(s) = 0 \\ 1 & \texttt{if } hd(s) > 0 \end{cases}$$
$$tl(sign(s)) = sign(tl(s))$$

Let us prove that $sign(s) = sign(sign(s))$. We consider the set $R = \{sign(s) \equiv sign(sign(s)) \mid s \text{ is a stream}\}$.

Checking that $hd(sign(s)) = hd(sign(sign(s)))$ can only be done by making a case analysis over $hd(s)$. If, for instance, $hd(s) < 0$, then $hd(sign(s)) = -1$. Therefore $hd(sign(s)) < 0$, so $hd(sign(sign(s))) = -1$. Both the left hand side and the right hand side of the initial equation are reduced to -1 in this case. The other two cases, $hd(s) = 0$ and $hd(s) > 0$, are handled similarly.

In the end, we need to prove that $tl(sign(s)) \equiv tl(sign(sign(s))) \in R$, which is equivalent to $sign(tl(s)) \equiv sign(sign(tl(s))) \in R$. The property holds because

$tl(s)$ is a stream s' and $sign(s') \equiv sign(sign(s')) \in R$ for any s'. By the coinduction principle, $sign(s) = sign(sign(s))$.

To automate case analysis for this situation, we "encapsulate" the definition of *sign* in a single syntactic construct, named *guarded equation*. The tool is able to extract from such equations the information it needs to perform case analysis.

The general goal of this paper is to present our approach to automating coinduction with case analysis, and more concretely to present our extension of CIRC with case analysis statement constructs, to describe our automated implementation of case analysis, and to prove the soundness of our technique and implementation. A secondary but equally important goal is to propose a generic technique to deal with extensions of coinductive proof systems[1], based on what we call equational interpolants. An equational interpolant is a new sentence of the form $\langle e, itp \rangle$, where e is an equation and itp is a set of equations; each of the equations involved in an interpolant can be conditional and can have its own quantifiers, which can be different from the others'. The intuition for $\langle e, itp \rangle$ is simple: e holds whenever itp holds. In other words, to prove $E \vdash e$ one can chose to instead prove $E \vdash itp$. This is somewhat similar to the homonimous notion in Craig interpolation, though the later also imposes restrictions over the signature of the interpolant; we here only take over the intuition that the interpolant interposes between the hypotheses and the task to prove.

Equational interpolants can be used in two ways: 1) to extend the ability of the prover to automatically find new lemmas by preserving the initial entailment relation, and 2) to extend the initial entailment relation in a consistent way with the specification enriched with new constructs. All previous extensions of circular coinduction consist of adding new types of statements together with new proof rules for them to the proof system. Interestingly, all these can be captured as special instances of a similar but more general process involving equational interpolants: the specific statements can be regarded as particular interpolants and the specific proof rules can be regarded as corresponding instances of general interpolant rules. Case analysis is no different. We borrow the general definition of a case statement from [5], but we here capture its semantics as a particular case of specification with equational interpolants. An immediate advantage of our new approach is that we can use case analysis in both ways mentioned above.

In short, the solution adopted in CIRC for automated case analysis is:

- enrich the specification syntax with new constructs, named *CIRC case statements*, for declaring enumerated sorts and guarded equations;
- transform the new constructs above uniformly into *annotated case sentences* (which can be regarded as interpolants);
- extend the coinduction proof engine with a new rule, [CaseAn]; and
- redesign the algorithm for automatic detection of special contexts.

Section 2 introduces notions and notations used throughout the paper, and the derivation rules of CIRC. Section 3 shows how the prover may be extended

[1] The technique appears to be more general, but we have not experimented with it outside our framework discussed here.

with new rules associated to interpolants. Section 4 presents the formal representation of case sentences and their corresponding entailment relation. Section 5 introduces the new syntactic constructs for specifying CIRC case statements. Subsections 5.1 and 5.2 describe how we extend the coinduction engine with a new rule for case analysis, and, how we improve the algorithm for detecting special contexts using case sentences, respectively. Section 5.3 shows how to write behavioral specifications and prove properties using CIRC.

2 Behavioral Specification and Circular Coinduction

An *algebraic specification* is a triple $\mathcal{E} = (S, \Sigma, E)$, where S is a set of *sorts*, Σ is a *many-sorted signature* and E is a set of conditional equations of the form $(\forall X)\, t = t'$ `if` *cond*, where $cond = (\bigwedge_{i \in I} u_i = v_i)$, I is a set of indexes; t, t', u_i, and v_i $(i \in I)$ are Σ-terms with variables in X. If $I = \emptyset$ then the equation is *unconditional* and may be written as $(\forall X)\, t = t'$.

A *Σ-context* C is a Σ-term with one occurrence of a distinguished variable $*{:}s$ of sort s. The context is written more explicitly as $C[*{:}s]$ instead of just C. When Σ is understood, a Σ-context may be referred to as a *context*. If $C[*{:}s]$ is a context of sort s' and t is a term of sort s, then $C[t]$ is the term of sort s' obtained by replacing t for $*{:}s$ in C. Consider an equation $e : (\forall X)\, t = t'$ `if` *cond*. By $C[e]$ we denote the equation $(\forall X \cup Y)\, C[t] = C[t']$ `if` *cond*, where Y is the set of non-star variables occurring in $C[*{:}s]$.

A *behavioral specification* (e.g., [10]) is a triple $\mathcal{B} = (S, (\Sigma, \Delta), E)$, where S, Σ and E are the sets composing an algebraic specification $\mathcal{E}$, and Δ is a set of Σ-contexts, called *derivatives*. A derivative in Δ is written as $\delta[*{:}h]$. The sorts S are split in two classes: *hidden sorts*, $H = \{h \mid \delta[*{:}h] \in \Delta\}$, and *visible sorts*, $V = S \setminus H$. A *Δ-context* is inductively defined as follows: 1) each $\delta[*{:}h] \in \Delta$ is a Δ-context; and 2) if $C[*{:}h']$ is a Δ-context and $\delta[*{:}h]$ is a term of sort h' from Δ, then $C[\delta[*{:}h]]$ is a Δ-context. A *Δ-experiment* is a Δ-context of visible sort.

If $\delta \in \Delta$ and e is an equation, then $\delta[e]$ is called a *derivative* of e. Given an entailment relation $\vdash$ over $\mathcal{E}$, the *behavioral entailment* relation is defined as follows: $\mathcal{B} \Vdash e$ iff $\mathcal{E} \vdash C[e]$ for each Δ-experiment C appropriate for the equation e. In this case, we say that $\mathcal{B}$ *behaviorally satisfies* e. For the streams defined in Section 1, the derivatives are $hd[*{:}Stream]$ and $tl[*{:}Stream]$. If $\mathcal{B} = (S, (\Sigma, \Delta), E)$, then we often write $\mathcal{B} \vdash e$ for $(S, \Sigma, E) \vdash e$ and $\mathcal{B} \cup F$ for $(S, (\Sigma, \Delta), E \cup F)$. We assume that $\vdash$ satisfies properties like reflexivity, monotonicity, transitivity, and Δ-congruence (see [10] for more details).

Circular coinduction [4,10] is a coinductive proving technique for behavioral properties which can be defined as a proof system (see [10]). To prevent the use of coinductive hypotheses in contextual reasoning, circular coinduction uses a *freezing* operator $\boxed{-} : s \to Frozen$, defined for each sort s; *Frozen* is a new sort. A *frozen equation* is an equation of the form $(\forall X)\ \boxed{t} = \boxed{t'}$ if *cond*.

CIRC implements a circular coinduction engine for the proof system given in [10] using a set of reduction rules of the form $(\mathcal{B}, \mathcal{F}, \mathcal{G}) \Rightarrow (\mathcal{B}, \mathcal{F}', \mathcal{G}')$, where $\mathcal{B}$ represents the behavioral specification, $\mathcal{F}$ is the set of coinductive hypotheses

(a set of frozen equations) and $\mathcal{G}$ is the current set of goals. An equational *goal* (proof obligation) is a conditional equation g of the form $(\forall X)t = t'$ `if` *cond*. For the sake of the presentation, the goals are also represented as frozen equations.

Here is a brief description of the reduction rules underlying CIRC:

[Done]: $(\mathcal{B}, \mathcal{F}, \emptyset) \Rightarrow \cdot$
Whenever the set of goals is empty, the system terminates with success.

[Reduce]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G})$ `if` $\mathcal{B} \cup \mathcal{F} \vdash \boxed{e}$
If the current goal is a $\vdash$–consequence of $\mathcal{B} \cup \mathcal{F}$ then $\boxed{e}$ is dropped.

[Derive]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow (\mathcal{B}, \mathcal{F} \cup \{\boxed{e}\}, \mathcal{G} \cup \{\boxed{\Delta[e]}\})$
`if` $\mathcal{B} \cup \mathcal{F} \not\vdash \boxed{e} \wedge e$ is hidden
When the current goal e is hidden and it is not a $\vdash$–consequence, it is added to the specification and its derivatives to the set of goals. $\boxed{\Delta[e]}$ denotes the set $\{\boxed{\delta[e]} \mid \delta \in \Delta\}$.

[Generalize]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{(\forall Y)\ \boxed{\theta(t)} = \boxed{\theta(t')}\}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{(\forall Y)\ \boxed{t} = \boxed{t'}\})$
where $\theta : X \to T_\Sigma(Y)$ is a substitution.
If the current goal can be generalized after identifying the substitution θ, then we replace it by its generalized form.

[Fail]: $(\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \{\boxed{e}\}) \Rightarrow$ *failure* `if` $\mathcal{B} \cup \mathcal{F} \not\vdash \boxed{e} \wedge e$ is visible
This rule stops the reduction process with failure whenever the current goal e is visible and cannot be proved using $\vdash$.

The entailment relation used in CIRC is $\vdash_{\rightarrow\leftarrow} : \mathcal{E} \vdash_{\rightarrow\leftarrow} (\forall X)t = t'$ `if` $\bigwedge_{i \in I} u_i = v_i$ iff $\mathrm{nf}(t) = \mathrm{nf}(t')$, where $\mathrm{nf}(t)$, the *normal form* of t, is computed using an enhanced version of the initial specification:

- the variables X of the equations are turned into fresh constants;
- the condition equalities $u_i = v_i$ are added as equations to the specification;
- the equations in the specification are oriented and used as rewrite rules on t.

The rules [Done], [Reduce], and [Derive] implement the proof rules with the same names given in [10]. The rule [Generalize] is presented in [6]. After a failing stop signaled by [Fail], further human intervention is required in order to identify the source of the failure. There are also some additional rules used only for optimization purposes. An example of such a rule is [Normalize], which computes the normal form of an equation and can be used, for instance, in combination with the rule [Derive].

The next result is a variant of the soundness theorem given in [10].

Theorem 1 (Soundness). *Let $\mathcal{B}$ be a behavioral specification, and $\vdash$ an entailment relation. If $(\mathcal{B}, \mathcal{F}_0 = \emptyset, \mathcal{G}_0 = \boxed{G}) \Rightarrow^* (\mathcal{B}, \mathcal{F}_n, \mathcal{G}_n = \emptyset)$, using [Reduce] and [Derive], then $\mathcal{B} \Vdash G$.*

We use this result in the next section. The proof of correctness for the rule [Generalize] is given in [6]; in the next section we show that it can be regarded as a particular case of a more general technique.

3 Extending CIRC with Equational Interpolants

In this section we present a technique that allows us to easily extend the proof system with new reduction rules.

Definition 1. 1) *If Σ is a signature then a Σ-*equational interpolant *is a pair $\langle e, itp\rangle$, where e is a Σ-equation and itp is a finite set of Σ-equations.*
2) *A* behavioral specification with interpolants $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ *is a behavioral specification $(S, (\Sigma, \Delta), E)$ together with a set $\mathcal{I}$ of interpolants. An entailment relation for E is extended to $(E, \mathcal{I})$ as follows: in the definition of $\vdash$ E is replaced with $(E, \mathcal{I})$ and a new rule is added:*

$$\frac{(E,\mathcal{I}) \vdash itp}{(E,\mathcal{I}) \vdash e} \text{ if } \langle e, itp\rangle \in \mathcal{I} \tag{1}$$

3) *If $\vdash$ is an entailment relation for E and $\mathcal{I}$ is a set of interpolants, then we say that $\mathcal{I}$ is* $\vdash$–preserving *if $E \vdash itp$ implies $E \vdash e$, for each $\langle e, itp\rangle \in \mathcal{I}$.*

Theorem 2. *Let $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{I}))$ and $\vdash$ be an entailment relation such that $\mathcal{I}$ is $\vdash$–preserving. If e is a Σ–equation then $(S, (\Sigma, \Delta), E) \Vdash e$ if and only if $(S, (\Sigma, \Delta), (E, \mathcal{I})) \Vdash e$.*

The CIRC engine associates a rewrite rule of the form:

$$[\mathsf{itp}]\text{: } (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \boxed{e}) \Rightarrow (\mathcal{B}, \mathcal{F}, \mathcal{G} \cup \boxed{itp})$$

with each interpolant $\langle e, itp\rangle \in \mathcal{I}$. We write $[\mathsf{itp}] \in \mathcal{I}$ in order to denote that the rule $[\mathsf{itp}]$ is associated with an interpolant from $\mathcal{I}$. The following theorem states that if we enhance the proof system presented in [10] with interpolants, then it remains sound w.r.t. the new entailment relation.

Theorem 3. *Let $\mathcal{B}$ be a behavioral specification, $\vdash$ an entailment relation, and $\mathcal{I}$ a set of interpolants. If $(\mathcal{B}, \mathcal{F}_0 = \emptyset, \mathcal{G}_0 = \boxed{G}) \Rightarrow^* (\mathcal{B}, \mathcal{F}_n, \mathcal{G}_n = \emptyset)$, using [Reduce], [Derive] and the rules associated to $\mathcal{I}$, then $\mathcal{B} \Vdash G$.*

A consequence of the proof of Theorem 3 is that the equational interpolants preserve the circular coinduction principle [10]:

Corollary 1. *In the hypothesis of Theorem 3, there is a set of frozen equations $\boxed{F}$ such that $\mathcal{B} \cup \boxed{F} \vdash \boxed{\Delta[F]}$.*

Notice that in the conclusion of Theorem 3, $\Vdash$ is built on the extended $\vdash$ (the extension of the initial entailment to specifications with interpolants). If $\mathcal{I}$ is $\vdash$-preserving, then $\Vdash$ is the behavioral extension of the initial $\vdash$. Therefore there are two ways of using interpolants within a coinductive proof:

Implicit use. In this case the equational interpolants must be $\vdash$–preserving and are used to justify why other rules are sound; the soundness of their use is given by Theorem 2. An example of implicit using of equational interpolants is that of the generalization rule [6]. By this rule, a concrete equation $u = u'$ is replaced

by a more general one $t = t'$. This means that there is a substitution θ such that $\theta(t) = u$ and $\theta(t') = u'$. It is easy to see that [Generalize] for $u = u'$ and θ is equivalent with the rule [itp] corresponding to the interpolant $\langle u = u', \{t = t'\}\rangle$. Theorem 4 in [6] becomes a direct consequence of Theorem 3.

Explicit use. The specification includes special syntactical constructs for denoting equational interpolants. For instance, the equational interpolants may be explicitly included in a CIRC theory using simplification rules [6] or case sentences (see Section 5). In this case the user is the one who decides whether the equational interpolants included in the specification are $\vdash$–preserving or that they really extend $\vdash$. In other words, the explicit use of equational interpolants is a part of the specification design.

4 Specifications with Cases

In this section we recall from [5] the definitions for case sentences and we show that their semantics can be given by means of equational interpolants.

Let (S, Σ) be an algebraic signature. A Σ-*case sentence* over the set of variables Y is a nonempty set $\{case_i \mid i \in I\}$ written as $(\forall Y)\,(\bigvee_{i\in I} case_i)$, where $case_i = (\bigwedge_{j\in J_i} u^i_j = v^i_j)$, and $u^i_j, v^i_j \in \mathcal{T}_\Sigma(Y)$, for each $i \in I$ and $j \in J_i$. Hence, cases and conditions have the same syntax. If $\theta : Y \to T_\Sigma(X)$ is a substitution, then $\theta(case_i)$ denotes the case $\theta(\bigwedge_{j\in J_i} u^i_j = v^i_j) = \bigwedge_{j\in J_i} \theta(u^i_j) = \theta(v^i_j)$. A *specification with cases* is a triple $\mathcal{E} = (S, \Sigma, (E, \mathcal{C}))$, where (S, Σ, E) is an algebraic specification and $\mathcal{C}$ is a set of case sentences.

An entailment relation $(S, \Sigma, E) \vdash e$ can be extended to specifications with cases $(S, \Sigma, (E, \mathcal{C})) \vdash e$ by means of equational interpolants. Each specification with cases $(S, \Sigma, (E, \mathcal{C}))$ is associated with a specification with equational interpolants $(S, \Sigma, (E, \mathcal{I}_\mathcal{C}))$, where $\mathcal{I}_\mathcal{C}$ is the set of pairs $\langle e, itp_{case,\theta}(e)\rangle$ with e an equation $(\forall X)t = t'$ if *cond*, *case* a case sentence $(\forall Y)(\bigvee_{i\in I} case_i)$ in $\mathcal{C}$, $\theta : Y \to T_\Sigma(X)$ a substitution, and $itp_{case,\theta}(e)$ the set $\{(\forall X)t = t' \text{ if } cond \wedge \theta(case_i) \mid i \in I\}$. In other words, each triple that consists of an equation, a case sentence, and a substitution uniquely defines an equational interpolant. The second inference rule from Definition 1 interpreted for the interpolant defined by a case sentence becomes similar to the one given in [5]:

$$\frac{\begin{array}{l}(\forall Y)\,(\bigvee_{i\in I} case_i) \in \mathcal{C},\\ \theta : Y \to \mathcal{T}_\Sigma(X),\\ (\forall i \in I)(S, \Sigma, (E, \mathcal{C})) \vdash (\forall X)\,t = t' \text{ if } cond \;\wedge\; \theta(case_i)\end{array}}{(S, \Sigma, (E, \mathcal{C})) \vdash (\forall X)\,t = t' \text{ if } cond} \tag{2}$$

However, we do not impose a "completeness" condition for case sentences, which in our terms is equivalent to saying that the interpolants $\mathcal{I}_\mathcal{C}$ are $\vdash$–preserving. For instance, we may have a specification as follows:

$$\begin{array}{ll} even(0) = true & f(x) = x \text{ if } even(x)\\ even(s(0)) = false & f(x) = s(x) \text{ if } not\ even(x)\\ even(s(s(x))) = even(x) & \end{array}$$

If we consider the case sentence c given by $(\forall x) even(x) = true \vee even(x) = false$ and $\vdash$ denotes the equational deduction, then we have $(S, \Sigma, (E, \{c\})) \vdash (\forall x) even(f(x)) = true$. Obviously, we cannot infer from the specification that $(S, \Sigma, E) \vdash (\forall x)(even(x) == true$ or $even(x) == false)$ (and hence $(S, \Sigma, E) \not\vdash (\forall x) even(f(x)) = true$) because this property is an inductive consequence. Therefore the rule (2) is a real extension of the initial entailment relation (here the equational deduction). The explicit use of case sentences in specifications may directly influence the definition of the entailment relation. In this way the user has a larger freedom in using case analysis.

As it is noted in [5], the use of case analysis at this level of generality is very expensive and finding an appropriate substitution θ is a difficult task which cannot be easily automatized. In [4] the following method is proposed: each case sentence comes with a pattern, usually denoted by p, which is just a Σ-term with variables. The case analysis rule is enabled only if the pattern p matches a subterm of t or t', and then the substitution also comes for free. The user must specify the pair (pattern, case sentence) to be used for a particular task.

We want to exploit the same idea but in an automated way. For this, we introduce *annotated case sentences*, which are pairs of the form $(p, \{case_i \mid i \in I\})$. Here, $p \in T_\Sigma(Y)$ is the *pattern* and $(\forall Y) (\bigvee_{i \in I} case_i)$ is a case sentence.

As we said above, the case analysis is a proper component of the specification. Therefore the solution we propose here is to include in the specification language special syntactical constructs from which annotated case sentences can be automatically computed. Knowing the set of annotated case sentences included in the specification, a prover can supervise the case analysis making use of proof tactics. In the next section we introduce three such syntactical constructs.

5 Implementation in CIRC

CIRC theories extend the syntax of Full-Maude theories by allowing the user to specify *derivatives* [7], *special contexts* [8] and *simplification rules* [6]. Here we present how one can use new syntactic constructs in CIRC theories that enable the prover to automatically use case analysis. These new syntactic constructs are named *CIRC case statements*. We introduce three types of CIRC case statements: enumerated sorts, guarded equations and annotated case sentences.

Enumerated sorts are declared using the syntax "`enum` s `is` $ct_1 \ \ldots \ ct_n$.", where s is the name of the sort and $ct_i, i = \overline{1..n}$, are the constants that define it. The *guarded equations* syntax is: "`geq` $t = \ t_1$ `if` $case_1$ `[]` $\ldots$ t_n `if` $case_n$ `[]` .", where t and $t_i, i = \overline{1..n}$ are $\mathcal{T}_\Sigma(Y)$-terms and $case_i, i = \overline{1..n}$ are disjoint conditions. The notation for guarded equations is inspired from Dijkstra's command language [3], except that the syntax of the guards is inspired from the Maude convention for the conditional equations. *Annotated case sentences* are directly declared using the syntax "`cases pattern` $= p$ `if` $case_1$ `\/` $\ldots$ `\/` $case_n$.".

The syntactic constructs presented above are not disjoint. For instance, the enumerated sorts and the guarded equations can be seen as particular instances of the annotated case sentence (see below). We found these syntactic constructs adequate in most of the case studies we considered. We believe they are more

intuitive and familiar to a programmer than a syntactic construct that allows the direct definition of an annotated case sentence. Moreover, the solution we present here can easily be extended with other syntactic constructs if they are considered to be useful in practice.

A *CIRC specification with cases* is a triple $\mathcal{E} = ((S, S^e), \Sigma, (E, E^g, \mathcal{C}))$, where S, Σ, E have similar meanings to those from the equational many sorted specifications, S^e is a set of enumerated sorts, E^g is a set of guarded equations, and $\mathcal{C}$ are the annotated case sentences. We associate each $\mathcal{E} = ((S, S^e), \Sigma, (E, E^g, \mathcal{C}))$ with a "compiled" specification with cases $\widetilde{\mathcal{E}} = (\widetilde{S}, \widetilde{\Sigma}, (\widetilde{E}, \widetilde{\mathcal{C}}))$, where:

- $\widetilde{S}$ is S together with the names of the enumerated sorts;
- $\widetilde{\Sigma}$ is Σ together with the constants of the enumerated sorts;
- $\widetilde{E}$ is E together with the conditional equations $(\forall Y)t = t_i$ if $case_i$, $i = \overline{1..n}$, for each guarded equation in E^g;
- $\widetilde{\mathcal{C}}$ is the set of annotated case sentences obtained in the following manner:
  - any enumerated sort "`enum` s `is` ct_1 ... ct_n ." defines the annotated case sentence $(\forall\{y\})\,(y, y = ct_1 \vee \cdots \vee y = ct_n)$, where y is of sort s;
  - any guarded equation "`geq` $t = t_1$ `if` $case_1$ `[]` ... t_n `if` $case_n$ `[]` ." defines the annotated case sentence $(\forall Y)\,(t, case_1 \vee \ldots \vee case_n)$, where Y is the set of the variables occurring in the guarded equation;
  - any sentence "`cases pattern` $= p$ `if` $case_1$ `\/` ...`\/` $case_n$." defines the annotated sentence $(\forall Y)\,(p, case_1 \vee \ldots \vee case_n)$, where Y is the set of the variables occurring in the pattern p.

Example 3. For the first example presented in Section 1 we have one enumerated sort in S^e: `enum Bit is 0 1 .` $\widetilde{\mathcal{E}}$ has the following components:

- $\widetilde{S} = S \cup \{\texttt{Bit}\}$;
- $\widetilde{E} = E$;
- $\widetilde{\Sigma} = \Sigma \cup \{$`op 0 : -> Bit .` , `op 1 : -> Bit .`$\}$;
- $\widetilde{\mathcal{C}} = \{(\texttt{B},\ \texttt{B} = \texttt{0} \vee \texttt{B} = \texttt{1})\}$, where `B` is a variable of sort `Bit`.

Example 4. For the second example, we have one guarded equation in E^g:

```
geq hd(sign(S)) =
     -1 if hd(S) < 0 = true []
      0 if hd(S) = 0        []
      1 if hd(S) > 0 = true [] .,
```

where `S` is a variable of sort `Stream`. $\widetilde{\mathcal{E}}$ is given by:

- $\widetilde{S} = S$; $\widetilde{\Sigma} = \Sigma$;
- $\widetilde{E} = E \cup \{$ `ceq hd(sign(S)) = -1 if hd(S) < 0 = true .` ,
  `ceq hd(sign(S)) =  0 if hd(S) = 0 .` ,
  `ceq hd(sign(S)) =  1 if hd(S) > 0 = true .` $\}$;
- $\widetilde{\mathcal{C}} = \{(\texttt{hd(sign(S))},\ \texttt{hd(S)} < \texttt{0} = \texttt{true} \vee \texttt{hd(S)} = \texttt{0} \vee \texttt{hd(S)} > \texttt{0} = \texttt{true})\}$.

Instead of using guarded equations, one could also declare the conditional equations from $\widetilde{E}$ presented above and specify the case sentence by: "`cases`

`pattern` = `hd(sign(S)) if hd(S) < 0 = true \/ hd(S) = 0 \/ hd(S) > 0 = true .`". In this way the user has the freedom to choose the syntax which describes his/her system in the best way.

It is worth noting that if a specification with cases is used, then the entailment relation is used during the proving process is that given by the associated equational interpolants (see Section 4).

5.1 Extending the Circular Coinduction Engine

In this section we describe how we enhanced the CIRC engine with automatic case analysis, and prove the correctness of our extension. We here only consider a behavioral specification with general cases, $\widetilde{\mathcal{B}} = (\widetilde{S}, (\widetilde{\Sigma}, \Delta), (\widetilde{E}, \widetilde{\mathcal{C}}))$; other specialized case statements can be desugared into general ones, as explained above.

We extend the coinduction proving engine with the reduction rule [CaseAn]. This rule replaces a conditional equation $t = t'$ `if` $cond$ by a set of equations $\{t = t' \texttt{ if } cond \wedge \theta(case_i) \mid i \in I\}$ if the case sentence $(\forall Y)(p, \bigvee_{i \in I} case_i)$ is in $\widetilde{\mathcal{C}}$ and $\theta(p)$ is a subterm of t or t'. [CaseAn] is therefore an instance of the rule [itp] corresponding to $\langle t = t' \texttt{ if } cond, \{t = t' \texttt{ if } cond \wedge \theta(case_i) \mid i \in I\}\rangle$:

$$\begin{aligned}\textsf{[CaseAn]} :& (\widetilde{\mathcal{B}}, \mathcal{F}, \mathcal{G} \cup \{\boxed{t} = \boxed{t'} \texttt{ if } cond\}) \Rightarrow \\ & (\widetilde{\mathcal{B}}, \mathcal{F}, \mathcal{G} \cup \{\boxed{t} = \boxed{t'} \texttt{ if } cond \wedge \theta(case_i) \mid i \in I\} \\ & \quad \text{if } (\forall Y)(p, \bigvee\nolimits_{i \in I} case_i) \text{ is in } \widetilde{\mathcal{C}} \text{ and } \theta(p) \text{ is a subterm of } t \text{ or } t'\end{aligned}$$

where $\mathcal{F}$ is the set of frozen axions and $\mathcal{G} \cup \{\boxed{t} = \boxed{t'} \texttt{ if } cond\}$ is the current set of goals. By Theorem 3, the extended engine is sound because [CaseAn] is a rule associated to the interpolant defined by (2).

One of the challenges we encountered was to find the best candidate for case analysis when more than one substitution could be applied. In our experiments, the strategy that gave the best results was to identify a subterm of either t or t' with the smallest height possible where at least one of the patterns p from $\widetilde{\mathcal{C}}$ could provide a substitution.

5.2 Computing Special Contexts Using Cases

In this subsection we give an intuition on what special contexts are with a few examples, and present the new algorithm for detecting special contexts using case analysis, as well as some required notions, and the result expressing the correctness of the algorithm. The formal background for special contexts is presented more in detail in [8].

We have seen in Section 2 that the frozen hypotheses cannot be used in contextual reasoning. However, there are contexts under which it is "safe" to use the frozen hypotheses. In [8] it is defined such a class of contexts, called special contexts. A context $\gamma[*{:}h]$ is called *special* if, by definition, for any experiment C for γ there is some term t such that $\mathcal{B} \vdash C[\gamma[*{:}h]] = t$ and each occurrence of $*{:}h$ in t appears in a subterm which is an experiment of depth smaller than or equal to that of C.

Example 5. Let us consider the operation f over streams of bits defined as: $f(0:s) = 1:f(s), f(1:s) = 0:0:f(f(f(s)))$. By making a case analysis we deduce that $f(*{:}Stream)$ is a special context (see below). Knowing this, CIRC is able to automatically prove that $f(0^\infty) = 1^\infty$ and $f(1^\infty) = 0^\infty$.

Not all contexts are special. Consider, for instance, the operation *odd* defined by $odd(a:b:s) = a:odd(s)$. Let s and t be specified by $hd(s) = hd(t)$, $tl(s) = odd(s)$ and $tl(tl(t)) = odd(t)$. If we wrongly assume that $odd(*{:}Stream)$ is special then we manage to prove that $odd(t) = s$, which is unsound.

CIRC has been able so far to automatically detect special contexts deriving from the operators defined using unconditional equations. Now the prover may detect contexts that derive from operators defined using specifications with cases (and implicitly, conditional equations). For instance, with the enhanced algorithm, CIRC detects that $sign(*{:}Stream)$, introduced in the second motivating example, and $f(*{:}Stream)$, defined in Example 5, are special contexts.

We next present in detail the algorithm computing special contexts for specifications with case sentences and its correctness. In order to fix the terms of the discussion, for the rest of this subsection we consider the following items:

- $\mathcal{B} = ((S, S^e), (\Sigma, \Delta), (E, E^g, \mathcal{C}))$, a fixed CIRC confluent and terminating behavioral specification with cases;
- $\widetilde{\mathcal{B}} = (\widetilde{S}, (\widetilde{\Sigma}, \Delta), (\widetilde{E}, \widetilde{\mathcal{C}}))$, the compiled behavioral specification with cases;
- any derivative $\delta \in \Delta$ is $(\widetilde{S}, \widetilde{\Sigma}, \widetilde{E})$-irreducible;
- $Ctx(\Sigma)$, the set of all Σ-contexts
- $\Sigma^{hidden} \subseteq \widetilde{\Sigma}$, the set of operations with hidden result and at least one hidden argument;
- $Ctx^\circ(\Sigma^{hidden})$, the set of contexts $f(x_1, \ldots, x_n)$ with $f \in \Sigma^{hidden}$, $x_i = *$ for exactly one hidden argument i and for $j \neq i, x_j$ are variables;
- for a Δ-context C, the *hidden depth* of C is defined by $|C|^\bullet = |C|$ if C is hidden, and $|C|^\bullet = |C| - 1$ if C is visible;
- a fixed set $\Gamma \subseteq Ctx^\circ(\Sigma^{hidden})$
- a *generalized constant* is an operation whose arguments are of visible sort

The problem of deciding if a given context is special for a given specification is Π^0_2-complete. This complexity is due to the facts that $\mathcal{B} \vdash C[\gamma[*{:}h]] = t$ must be tested for all Δ-experiments C and that testing $C[\gamma[*{:}h]] = t$ with t satisfying the property from the definition of the special contexts is recursive enumerable. In practice, t is the normal form of $C[\gamma[*{:}h]]$. Moreover, the set of candidates for special contexts is also infinite. So, the best we can do is to find an algorithm which tests a property similar to the one above for a finite set of candidates and a finite set of Δ-contexts. Regarding the candidates, we are looking for a maximal set of minimal depth special contexts which is closed under composition: if γ_1 and γ_2 are special and $\gamma_1[\gamma_2]$ is defined, then $\gamma_1[\gamma_2]$ is special. The minimal set of Δ-context which must be tested is Δ itself. Therefore, as described in [8], we try to find a property $Comp(C, t)$ and a set $\Gamma \subseteq Ctx^\circ(\Sigma^{hidden})$ such that the property $Special(\Gamma)$, given by:

$$Special(\Gamma) \stackrel{\text{def}}{=} (\forall \gamma \in \Gamma)(\forall \delta \in \Delta)\, Comp(\delta, \gamma)$$

implies that each Γ-context is special. If we have an algorithm for computing the predicate $Comp(C, t)$, then the searching for a suitable Γ requires the evaluation of the predicate for a small set of pairs (C, t).

First we present an axiomatic definition for the predicate *Comp*.

Definition 2. *A* Δ-compositional structure *for* Γ *is a pair* $(\mathcal{T}, Comp)$*, where* $\mathcal{T}$ *is a set of terms and* $Comp(C, t)$ *is a predicate defined over* Δ*-contexts* C *and terms* $t \in \mathcal{T}$*, which together satisfy the following conditions:*

1. $Ctx(\Gamma) \subseteq \mathcal{T}$
2. $Comp(*, t) = Comp(t, *) = true$*;*
3. *let* C_1 *and* C_2 *be two* Δ*-contexts such that* $C_1[C_2]$ *is defined; if* $Comp(C_2, t_2)$ *and* $(\forall t \in \mathcal{T})\, Comp(C_1, t)$*, then* $Comp(C_1[C_2], t_2)$*;*
4. *let* γ_1 *and* γ_2 *be two* Γ*-contexts such that* $\gamma_1[\gamma_2]$ *is defined; if* $Comp(C, \gamma_1)$ *and* $((\forall D \in Ctx(\Delta))\ |D|^\bullet \leq |C|^\bullet$ *implies* $Comp(D, \gamma_2)$*, then* $Comp(C, \gamma_1[\gamma_2])$*.*

The first main result of this section shows that the property $Special(\Gamma)$ can be extended to Γ-contexts and Δ-contexts for the case of a Δ-compositional structure.

Theorem 4. *Let* $(\mathcal{T}, Comp)$ *a* Δ*-compositional structure for* Γ*. If* $Special(\Gamma)$*, then* $(\forall \gamma \in Ctx(\Gamma))(\forall C \in Ctx(\Delta))\, Comp(C, \gamma)$*.*

A particular structure $(\mathcal{T}, Comp)$ for specifications without cases is given in [8]. Here we extend that structure to specifications with cases. The definition for $\mathcal{T}$ remains unchanged, namely that of (k, Γ)-composite terms; we recall it in order to make the paper self-contained.

Definition 3. *Let* k *be an integer number* ≥ -1*. A* (k, Γ)-composite *is defined as follows:*

1. *any non-star variable and any constant is a* $(-1, \Gamma)$*-composite;*
2. *any* Δ*-context* C *is a* $(|C|^\bullet, \Gamma)$*-composite;*
3. *if* $f : v_1 \ldots v_n \to v$ *is a data operator or a generalized constant and* t_i *is a* (k_i, Γ)*-composite for* $i = 1, \ldots, n$*, then* $f(t_1, \ldots, t_n)$ *is a* (k, Γ)*-composite, where* $k = \max\{k_1, \ldots, k_n\}$*;*
4. *if* $\gamma \in \Gamma$ *and* t *is a* (k, Γ)*-composite, then* $\gamma[t]$ *is a* (k, Γ)*-composite;*
5. *if* C *is a* Δ*-context,* t *a* (k, Γ)*-composite with* $k = -1$ *or* t *of the form* $g(t_1, \ldots, t_n)$ *with* g *generalized constant, then* $C[t]$ *is a* (k, Γ)*-composite.*

The new definition of the predicate *Comp* is based on the notion of normal form of a term computed in the presence of cases.

We first define the operation *Eqn*, which transforms the provided conjunction of cases into a set of equations in the following manner:

$$Eqn(\bigwedge_{i \in I} u_i = v_i) = \bigcup_{i \in I} Eqn(u_i = v_i)$$

$$Eqn(u_i = v_i) = \begin{cases} \{u_i = v_i\}, & \text{the set variables from } u_i \text{ is included in that of } v_i \\ \{v_i = u_i\}, & \text{otherwise} \end{cases}$$

If $\widetilde{\mathcal{C}}$ is a set of annotated case sentences, then the $\widetilde{\mathcal{C}}$-*normal form* of a term t is the set $\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(t)$ of pairs $\langle t', case\rangle$ satisfying: $t' = \mathrm{nf}_{\widetilde{E}\cup Eqn(case)}$ and there is no pattern p in $\widetilde{\mathcal{C}}$ which is an instance of a subterm of t'. The component *case* is a conjunction of cases in $\widetilde{\mathcal{C}}$ used in the rewriting obtaining the irreducible term t'. An algorithm computing $\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}$ is:

$\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}} \leftarrow \langle \mathrm{nf}_{\widetilde{E}}(C[t]), nil\rangle$
`while` $(\exists\langle t', case\rangle \in \mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}})(\exists\theta : Y \to \mathcal{T}_\Sigma(X))(\exists(p, \bigvee_{i\in I} case_i) \in \widetilde{\mathcal{C}})$
  such that $\theta(p)$ is a subterm of t' `do`
  $\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}} \leftarrow \mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}} - \{\langle t', case\rangle\}$
    $\cup\ \{\langle \mathrm{nf}_{\widetilde{E}\cup Eqn(case)\cup Eqn(\theta(case_i))}(t'), case \wedge \theta(case_i)\rangle \mid i \in I\}$

Let $terms(\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(t))$ denote the set $\{t' \mid \langle t', case\rangle \in \mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(t)\}$. We can define now the predicate *Comp*:

$$Comp(C, t) \stackrel{\text{def}}{=} (\forall t' \in terms(\mathrm{nf}_{\widetilde{E},\widetilde{\mathcal{C}}}(C[t])))\ t' \text{ is a } (k', \Gamma)\text{-composite} \wedge k' \le k + |C|^\bullet$$

where t is a (k, Γ)-composite, and C is a Δ-context. Since any Γ-context is a $(0, \Gamma)$-composite, $C[*] = C$ and $*[t] = t$.

Theorem 5. *Let $\mathcal{B} = (S, (\Sigma, \Delta), (E, \mathcal{C}))$ be a behavioral specification with cases and Γ a subset of $Ctx^\circ(\Sigma^{hidden})$ such that $Special(\Gamma)$ holds. Then any Γ-context γ is special.*

Example 6. Consider the operation f defined in Example 5. The normal forms of $hd(f(s))$ and $tl(f(s))$ are $\{\langle 1, hd(s) = 0\rangle, \langle 0, hd(s) = 1\rangle\}$ and $\{\langle f(tl(s)), hd(s) = 0\rangle, \langle 0 : f^3(tl(s)), hd(s) = 1\rangle\}$ respectively. If $\Gamma = \{f(*{:}Stream)\}$, then it is easy to see that $Special(\Gamma)$ holds and therefore $f(*{:}Stream)$ is a special context.

Theorem 5 is the foundation for an algorithm computing a set of context $\Gamma \subseteq Ctx^\circ(\Sigma^{hidden})$, which is a basis for special contexts. The description of the algorithm is the same as the one presented in [8], except that the call $Comp(\delta, \gamma)$ requires the computation of the normal forms with cases for $\delta[\gamma]$, as presented above. Therefore the theorem above also ensures the correctness of both versions of the algorithm.

5.3 CIRC with Case Analysis at Work

In this section we present how CIRC theories are specified and a few commands in order to automatically prove properties using case analysis. The reader may use the web interface at `http://fsl.cs.uiuc.edu/index.php/Special:CircOnline` in order to test the examples.

Let us use CIRC in order to prove that by merging two infinite sorted streams of natural numbers we obtain a sorted stream. This is not a trivial example; even the proof by hand requires a significant effort. The sorted property can be defined by $isSorted(S) = hd(S) < hd(tl(S)) \wedge isSorted(tl(S))$. The merge operation

is defined by $hd(merge(S,S')) = hd(S)$ if $hd(S) < hd(S')$, $hd(merge(S,S')) = hd(S')$ if $hd(S) \geq hd(S')$, $tl(merge(S,S')) = merge(tl(S),S')$ if $hd(S) < hd(S')$, $tl(merge(S,S')) = merge(S,tl(S'))$ if $hd(S) \geq hd(S')$ and can be specified by two guarded equations. We further consider another operation, *toBits* that transforms the provided stream of natural numbers into a stream of bits in this manner: $hd(toBits(S)) = 1$ if $hd(S) < hd(tl(S))$, $hd(toBits(S)) = 0$ if $hd(S) \geq hd(tl(S))$, $tl(toBits(S)) = toBits(tl(S))$. The operation above can also be specified using guarded equations or two conditional equations together with a case sentence for the pattern *hd(S)*. If *ones* denotes the stream of 1's, then the property above is equivalent to:

$$toBits(merge(S_1{:}Stream, S_2{:}Stream)) = ones \text{ if } isSorted(S_1{:}Stream) = true \wedge isSorted(S_2{:}Stream) = true$$

Even though *merge* is defined using guarded equations, the algorithm succeeds to find that $merge(*{:}Stream, S{:}Stream)$ and $merge(S{:}Stream, *{:}Stream)$ are special contexts. The context $toBits(*{:}Stream)$ is not found because the definition of *toBits* does not fulfill the criteria checked by the algorithm (the definition of $hd(toBits(S))$ depends on a bigger experiment, $hd(tl(S)))$; this can be seen as a limitation of the algorithm. Recall that the problem of special contexts is Π_2^0-complete, so there is no an algorithm able to always find all special contexts.

We present the dialog needed to prove that by merging two sorted streams we obtain a sorted stream. After the tool and specification are loaded, three commands are need to prove this property:

- `(initialize .)`, which sets the initial state of the prover;
- add the property as an initial goal:
```
(add cgoal toBits(merge(S1:Stream,S2:Stream)) = ones
       if isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true .)
```
- `(coinduction .)`, which launches the circular coinduction engine.

Here is the full dialog with CIRC, where we can see that *merge* defines indeed special contexts.

```
> (initialize .)
Initializing ...
The special contexts are:
merge(*:Stream,V#2:Stream)
merge(V#1:Stream,*:Stream)

> (add cgoal toBits(merge(S1:Stream,S2:Stream)) = ones
           if isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true .)

> (coinduction .)
Proof succeeded.
  Number of derived goals: 10
  Number of proving steps performed: 39
  Maximum number of proving steps is set to: 256

Proved properties:
  toBits(merge(S1:Stream,S2:Stream)) = ones if
    isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true
```

The full proof for our property, given as inference rules, can be checked using the command `(show proof .)`. We present one of the rules in which we emphasize the application of [CaseAn]:

```
1. |||-  [* toBits(tl(merge(S1:Stream,S2:Stream))) *] = [* ones *] if
         isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true /\
         hd(S1:Stream)< hd(S2:Stream) = true
2. |||-  [* toBits(tl(merge(S1:Stream,S2:Stream))) *] = [* ones *] if
         isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true /\
         hd(S1:Stream)<= hd(S2:Stream) = false
---------------------------------------------------------- [Cases]
|||-  [* toBits(tl(merge(S1:Stream,S2:Stream))) *] = [* ones *] if
      isSorted(S1:Stream) = true /\ isSorted(S2:Stream) = true
```

Another challenging example is inspired from [9] and consists of proving that $Rev3(N)(Rev3(N)(S)) = S$, where

$$Rev3(N)(S) = Z3(T3(N)(S), T3(N-1)(S), T3(N-2)(S)$$

$$hd(Z3(S_1, S_2, S_3)) = hd(S_1) \qquad tl(Z3(S_1, S_2, S_3)) = Z3(S_2, S_3, tl(S_1))$$

$$hd(T3(N)(S)) = hd(tl^{n \bmod 3}(S)) \quad tl(T3(N)(S)) = T3(N)(tl^3(S))$$

with N ranging over natural numbers and S over streams. Even if the definition of $hd(T3(N)(S))$ is given by cases (because of $n \bmod 3$), is not recommended to use guarded equations for specifying it because it is possible to obtain patterns of the form $hd(T3(N-1)(S))$, which forces a case analysis on "$(N-1) \bmod 3$"; now we have to include in the specification how to compute "$(N-1) \bmod 3$" when we know "$N \bmod 3$" and how to compute "$N \bmod 3$" when we know "$(N-1) \bmod 3$" and this cannot be done using only rewriting (it is a source of non-termination). Therefore we specified it with conditional equations and we added the case sentence

$$\texttt{cases pattern} = N \texttt{ if } N \bmod 3 = 0 \vee N \bmod 3 = 1 \vee N \bmod 3 = 2\ .$$

Even so the proof is long and complex: 12 case analyses and 14 new lemmas automatically discovered.

6 Conclusions

We presented a simple and efficient solution for automating coinductive reasoning with case analysis. The starting point was the extension of the specifications and of the entailment relation with case sentences given in [5]. A novelty of the approach presented here consists of giving semantics to case sentences by means of equational interpolants, a general technique for extending coinductive provers like CIRC introduced also here. The soundness of the use of equational interpolants in the coinductive proving process is shown and hence the soundness of the reasoning with cases is obtained as a consequence.

The basic idea is to include special syntactical constructs in the specification language. These special constructs are then processed in order to exdd tract

annotated case sentences. A prover can supervise the application of the case analysis by means of proof tactics.

Using the concept of "normal form with cases", we were able to write algorithms and heuristics helping the prover. In particular, we showed that extending the algorithm computing the special contexts with case analysis, the prover was able to find a larger class of special contexts. Consequently, a larger class of properties can be proved. The simpler the predicate *Comp* is, the faster the algorithm for detecting special contexts becomes. Therefore, as future work, there is room and motivation for improving the form of the predicate.

Case analysis based on three syntactic constructs, enumerated sorts, guarded equations and annotated case sentences, has been implemented in CIRC and experiments showed that the new prover is able to handle a large class of practical examples.

A similar approach is given in [2], where the induction and a contextual simplification technique are used to prove behavioral (observational) properties. That approach also deals with case analysis and critical contexts (which are different from special contexts). Circular coinduction is more flexible because it is parametric in the basic entailment relation, and consequently can prove more coinductive properties. On the other hand, we currently do not disproof conjectures (CIRC only reports failure to find a proof under specified constraints, but not that the property is false).

Acknowledgment. The paper is supported in part by NSF grants CCF-0448501, CNS-0509321 and CNS-0720512, by NASA contract NNL08AA23C, CNCSIS grant PN-II-ID-393, and by ANCS 602/12516 (DAK).

References

1. Bouhoula, A., Rusinowitch, M.: Automatic case analysis in proof by induction. In: IJCAI, pp. 88–94. Morgan Kaufmann Publishers Inc., San Francisco (1993)
2. Bouhoula, A., Rusinowitch, M.: Observational proofs by rewriting. Theor. Comput. Sci. 275(1-2), 675–698 (2002)
3. Dijkstra, E.W.: Guarded commands, non-determinacy and formal derivation of programs. Commun. ACM 18(8), 453–457 (1975)
4. Goguen, J., Lin, K., Roşu, G.: Circular coinductive rewriting. In: ASE 2000: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, pp. 123–132. IEEE, Washington (2000)
5. Goguen, J., Lin, K., Roşu, G.: Conditional circular coinductive rewriting with case analysis. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) WADT 2003. LNCS, vol. 2755, pp. 216–232. Springer, Heidelberg (2003)
6. Goriac, E., Caltais, G., Lucanu, D.: Simplification and Generalization in CIRC. In: 12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE Computer Society, Los Alamitos (2009)
7. Lucanu, D., Goriac, E.-I., Caltais, G., Roşu, G.: CIRC: A behavioral verification tool based on circular coinduction. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 433–442. Springer, Heidelberg (2009)

8. Lucanu, D., Roşu, G.: Circular coinduction with special contexts. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 639–659. Springer, Heidelberg (2009)
9. Niqui, M., Rutten, J.J.M.M.: Sampling, splitting and merging in coinductive stream calculus. In: Mathematics of Program Construction 2010 (MPC 2010) (to appear, 2010); See CWI Technical report SEN-E0904 (2009) `http://homepages.cwi.nl/tilde~janr/papers/`
10. Roşu, G., Lucanu, D.: Circular Coinduction – A Proof Theoretical Foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 127–144. Springer, Heidelberg (2009)
11. Rutten, J.J.M.M.: A coinductive calculus of streams. Mathematical Structures in Computer Science 15(1), 93–147 (2005)

Enhanced Semantic Access to Formal Software Models

Hai H. Wang[1], Danica Damljanovic[2], and Jing Sun[3]

[1] School of Engineering and Applied Science, Aston University
H.WANG10@aston.ac.uk
[2] Department of Computer Science, University of Sheffield
D.Damljanovic@dcs.shef.ac.uk
[3] Department of Computer Science, The University of Auckland, New Zealand
j.sun@cs.auckland.ac.nz

Abstract. The success of the Semantic Web, as the next generation of Web technology, can have profound impact on the environment for formal software development. It allows both the software engineers and machines to understand the content of formal models and supports more effective software design in terms of understanding, sharing and reusing in a distributed manner. To realise the full potential of the Semantic Web in formal software development, effectively creating proper semantic metadata for formal software models and their related software artefacts is crucial. In this paper, a methodology with tool support is proposed to automatically derive ontological metadata from formal software models and semantically describe them.

Keywords: Semantic Web, OWL, Formal Methods, Z/Object-Z.

1 Introduction

Formal methods are defined as mathematically based techniques for the specification, development and verification of software and hardware systems[1]. The well-defined semantics and syntax of formal specification languages make them suitable for precisely capturing and formally verifying system requirements. Many formal and semi-formal specification techniques coexist and bring different advantages to system designers. For a complex system, normally more than one modelling approaches are used cooperatively in the process of the system development. This is due to the following reasons:

- Different software specification languages distinguish between each other in terms of the level of formality and usability.
- Different specification languages are designed for modelling different aspects of software systems.
- Different specification languages differ from each other in terms of the expressiveness and the level of tool support.
- Even for the same modelling language, different models may be developed during the life-cycle of a software system.

[1] http://en.wikipedia.org/wiki/Formal_methods

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 237–252, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Due to the complex mathematical nature of the formal methods and the lack of tool support, many software engineers have found that it is difficult to understand, incorporate and use different formal models consistently in the process of software developments, especially for large and complex systems. It is highly desirable to have different models and their related software artefacts systematically connected and collaboratively used, rather than in isolation. For a software system, the knowledge of its application domain, its different models and the related software artefacts should be sharable, linked and reusable in a consistent manner. However, the challenge is to have an open and flexible environment to support such a cooperation.

With the advent of the World Wide Web (WWW), its wide accessibility and open distributed nature have provided an important infrastructure for a promising environment for formal software specification and design. By using the Internet, software engineers can share, search, reuse and collaboratively develop software models more effectively. Formal methods such as the CafeOBJ system [9] has included an environment for supporting formal specification over the Internet. Others [4, 5, 17, 18] proposed to use Web browsers to display and navigate formal Z [16] models. Although the current Web environment has been successful in presenting information on the Internet, the lack of content information and the overburdened use of the display tags have made the efficient retrieval and exchange of information content difficult. The Semantic Web [14], emerged as the next generation of the Web, proposed the idea of having data on the Web defined and linked in a way that it can be understood by machines and used for automation and integration. This is achieved primarily by *annotating* information on the Web using semantic *metadata*. The success of the Semantic Web can have profound impact on the Web environment for formal specifications. For example, Dong et al. [7] proposed to use to use RDF [13] and DAML [19] to build a Semantic Web environment for extending and integrating various formal specification languages.

By using the semantic technology, we can potentially transform the formal software models and their domain application documentations into a conceptually organised and semantically interlinked knowledge space that incorporates data from multiple software artefacts produced during the process of the software development, e.g., forum posting, requirement documents, source code, configuration files, database, etc. The semantically enriched information can then be used to add novel functionalities to web-based documentation of the software concerned, providing the software engineers with new and powerful ways to comprehend and reuse software models. The realisation of this vision mainly depends on the fact whether we could effectively create proper *metadata* to semantically annotate formal software models. Without a systematic methodology and proper tool support, annotating formal software models could be a very tedious and expensive process, which carries a definite risk of failure. There is an urgent need to provide strategies and tools that allow the users to automatically or semi-automatically annotate formal models.

In this paper, we present a methodology and a prototype which has been developed in the course of the Transitioning Applications to Ontologies (TAO) project (`www.tao-project.eu`) to derive ontological metadata from formal software models and semantically describe them. Firstly, a framework that allows different formal

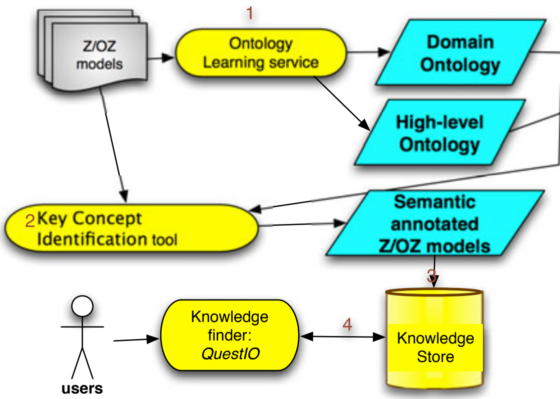


Fig. 1. Annotation Framework

software models to be described ontologically is defined. We then propose a methodology for interlinking the knowledge about the formal models, the application domain and other related software artefacts by semantically *annotating* them. We aim to automate the mechanical tasks during the annotating process, which can be divided into several major phases. Figure 1 shows the main steps of our approach. The ontology learning tool is used to derive ontologies from formal models (e.g., Z/Object-Z models) and other documents related to the application domain being modelled (e.g., specifications, UML diagrams, code documentations, software manuals, images, etc.). The content augment tool (KCIT) automatically identifies the key concepts within the software models and related documents, and annotates them using the domain ontology concepts. The distributed heterogeneous knowledge repositories are developed to efficiently index, query, and retrieve model contents, domain ontology and semantic annotations. An Integrated Development Environment (IDE) is developed to provide a coherent annotation support for users. In this paper, we use the formal Z/Object-Z(OZ) notations to illustrate the approach. Z is a formal specification language based on set theory and predicate logic [20]. It has been widely used in both industry and academic research for the specification and verification of software systems. Object-Z is an object-oriented extension of Z [8, 15]. TAO (`www.tao-project.eu`) is a project in the European Sixth Framework Program, with the goal to define methods and tools for transition of legacy information systems to semantic enabled services, enabling semantic interoperability between heterogeneous data resources and distributed applications.

The remainder of this paper is organised as follows. Section 2 briefly introduces the background material in the area of the Semantic Web. Section 3 presents how to automatically generate ontology metadata for representing the knowledge about the formal model and its application domain. Section 4 shows how the derived ontology can be used to create semantically augmented formal software models and related artefacts. Section 5 concludes the paper and discusses the future work.

2 Semantic Web

The Semantic Web is a vision for the next generation of Web with enhanced functionality that requires semantic-based representation and processing of Web information. The the World Wide Web Consortium (W3C) has proposed a series of technologies that can be applied to achieve this vision. The Semantic Web extends the current Web by giving the web content a well-defined meaning, better enabling computers and people to work in cooperation. XML is aimed at delivering data to systems with pre-agreed formats that can be understood and interpret by different programs. XML is focused on the syntax (defined by the XML schema or DTD) of a document and it provides essentially a mechanism to declare and use simple structured data. However there is no means for a program to actually understand the knowledge contained in the XML documents. Resource Description Framework (RDF) [13] is a foundation for processing metadata; which provides interoperability between applications that exchange machine-understandable information on the Web. RDF uses XML to exchange descriptions of Web resources and emphasizes facilities to enable automated processing. The RDF descriptions provide a simple ontology system to support the exchange of knowledge and semantic information on the Web. RDF Schema [6] provides the basic vocabulary to describe RDF documents. RDF Schema can be used to define properties and types of the web resources. The advent of RDF Schema represented an early attempt at an Semantic Web ontology language based on RDF.

OWL [2] is the latest standard in ontology languages, which was developed by members of W3C and the Description Logic (DL) community. An OWL ontology consists of classes, properties and individuals. *Classes* are interpreted as sets of objects that represent the individuals in the domain of discourse. Properties are binary relations that link individuals, which are represented as sets of ordered pairs that are subsets the cross product of the set of objects. OWL classes fall into two main categories – named classes and anonymous (unnamed) classes. Anonymous (unnamed) classes are formed from logical descriptions. They contain the individuals that satisfy the logical description. Anonymous classes may be sub-divided into *restrictions* and 'logical class expressions'. Restrictions act along properties, describing sets of individuals in terms of the types of relationships that the individuals participate in. For example, existential restrictions describe the individuals that have at least (the existence of) one relationship to individuals that are members of some other specified class. The existential restriction $\exists$ hasTopping PizzaTopping describes the set of individuals that have at least one hasTopping relationship to an individual that is a member of the class PizzaTopping. Cardinality restrictions describe sets of individuals in terms of the number of relationships that the individuals must participate in for a given property. There are three types of cardinality restriction – Minimum, Maximum and Exact. Logical classes are constructed from other classes using the boolean operators AND ($\sqcap$), OR ($\sqcup$) and NOT ($\neg$).

3 Ontology Representation of Software Models

An ontology is commonly defined as an explicit, formal specification of shared conceptualisation of a domain of interest. In order to establish a framework within which

formal software models can be developed, annotated and shared, creating standard ontological metadata for the formal software development is vital. The metadata can be divided into two categories of descriptions:

– the formal model itself and
– the application domain that the formal model specifies.

In this section we first present a Semantic Web environment for representing software models and follow by a method to generate domain ontology automatically from formal models and their related software artefacts. The ontology is used to interconnect different software artefacts by semantically annotating them.

To better illustrate the idea, a formal OZ model for WSMO Amazon Associates Web service (A2S) is used as the case study in this paper. Amazon Web services provide developers with direct access to Amazon technology platform. By using them, external developers and businesses can build their own applications on A2S in a reliable, flexible, and cost-effective manner. WSMO Amazon Web Services framework allows users to use the latest technology – Web Service Modelling Ontology (WSMO) to access Amazon Web Services[11]. WSMO is an enabling framework for the total/partial automation of the tasks (e.g., discovery, selection, composition, mediation, execution, monitoring, etc.) involved in both intra- and inter-enterprise integration of Web Services. To support the standardisation and tool usage of WSMO Amazon framework, a formal OZ model has been developed. This OZ model provides a rigorous formal semantics for WSMO and A2S, where the abstract syntax and static/dynamic semantics for each the WSMO and Amazon A2S construct are grouped together and captured in OZ classes; hence the language model is structural, concise and easily extensible. This OZ specification provides an invaluable adjunct to the current documentation and specifications, and supports further development, validation and verification of WSMO and its Amazon application as it evolves. For example, the following OZ class defines the WSMO $service\ element\ Capability$, which defines the functionality of a Web service.

$Capability$

$ServiceElement$

$importsOntology : \mathbb{P}\ Ontology$
$usesMediator : \mathbb{P}(ooMediator \cup WGMediator)$
$hasNonFunctionProperty : \mathbb{P}\ NonFunctionalProperty$
$hasSharedVariable : \mathbb{P}\ Variable$
$hasPrecondition, hasPostcondition : \mathbb{P} \downarrow Expression$
$hasAssumption, hasEffect : \mathbb{P} \downarrow Expression$
……

……

$amazonWSCapability : Capability$

……

A Web service capability is defined by specifying the *precondition*, *postcondition*, *assumption*, and *effect*, each of which is a set of *expressions*. A Web service capability also declares a set of variables shared between expressions. The terms used in these expressions must be formally defined in domain ontologies that must be imported either directly or via *ooMediators*. A *capability*, may be linked to certain goals that can be resolved by the Web service via special types of mediators, named *wgMediators*. The *amazonWSCapability* models the capability of Amazon Web services. The complete formal model can be found at the report [2].

However despite the advantages brought by this rather complicated OZ model, some software engineers find difficulties to understand and use it. Users have to refer to various knowledge about the WSMO and Amazon A2S, which are documented at different parts by over 30 documents. These documents are in different formats, ranging from plain English text, program code to XML definitions. Semantically interconnecting these knowledge is highly desirable.

3.1 Ontology for Software Specification

In order to provide a standard and machine processable interchange format for software models within the Semantic Web context, we first define an OWL ontological framework for describing formal software models (OWL-M). There are many advantages to represent formal models in the Semantic Web environment.

- There is a global naming schema – Uniform Resource Identifier (URI). The Semantic Web is generally built on syntax that use URIs to present data. URI provides a unified scheme to enable the cross-referencing and linking among different software model and model components.
- The RDF data model is used to represent information, where the information can be mapped directly and unambiguously to a decentralised model. Furthermore, in RDF there is no syntax constraint on the information representation. Everything is represented as through logical triples (i.e., *Subject*, *Predicate*, *Object*). Using RDF to represent software models, the information can be manipulated and the different model information can be combined, transferred and extracted easily. This means that the semantic data and syntax data can be distincted.
- The rich expressiveness of OWL allows us to set up the Semantic Web environments for different kinds of specification languages and translate them interchangeably. OWL has formal mathematical foundations in Description Logics, which help us to represented software models in a systematic and consistent way.
- The vision of the Semantic Web is to make the Web information computer-interpretable, thus enabling automation of many tasks currently performed by humans. Therefore, under our Semantic Web framework, many software modelling and design activities, like model transformation, refinement, composition etc., can be potentially automatically or semi-automatically performed by machines.

Figure 2 presents an overview of the top level of OWL-M. The OWL class *Specification* provides a reference point for a declared software model. One instance

[2] http://www-users.aston.ac.uk/~wangh10/report.pdf

of *specification* exists for each distinctly published software model. Each model instance relates to a *SpecificationProfile* that shows what application the specification models. It provides other meta information about the specification, and relates to a *SpecificationModel* that describes the specification content. Note that the Z/OZ notations are used as the example of modelling languages to illustrate the approach.

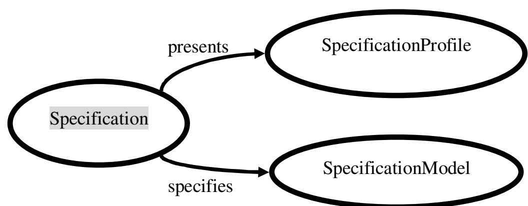


Fig. 2. Top level of the model ontology

Specification profile. It provides basic information about a software specification. This descriptive information may include non-functional meta-information and provenance. The specification profile can be used for a model-seeking agent to determine whether the specification meets its needs and also enabling software model sharing and managing. The class *SpecificationProfile* is a superclass of every type of high-level description. It is related to the class *Specification* by the property *presents* and its inverse property *presentedBy*.

The following shows some of the properties defined within the specification profile. We intentionally identify a set of application-domain/specification-language independent properties only. This model could be extended if needed (by defining suitable subclasses).

Model Name: It refers to the name of the specification model, which could be used as an identifier of the specification model.

Model Description: It provides a text description of the system being modelled.

Model Author: Models could have single or multiple authors, which could be people or organizations.

Service Contact Information: It contains the contact information, such as an email address, for people or agents requiring more information about the model.

Version: It provides the version of the model and date of creation.

Creation Time: It contains a date of the release of the specification. Agents can use it to check if a specification has been defined for the updated application requirements.

Specification Language: It provides the name of the specification language used for modelling, such as Z, VDM or UML.

The following OWL definition shows the OWL-M profile definition for the OZ model of the WSMO Amazon Web Services. *AmazonOZ* is defined as an instance of *OWLM*:*Specification*, where *OWLM* is the namespace for the software ontology proposed. It has *AmazonOZProfile* containing the profile information and

AmazonOZmodel containing the model itself (defined in the next subsection). The DL syntax for OWL is used here.

```
Individual(AmazonOZ  type(OWLM:Specification)
      value(presents  AmazonOZProfile)
      value(specifies AmazonOZmodel))
Individual(AmazonOZProfile  type(OWLM:SpecificationProfile)
      value(hasModelName
             "The Object-Z model for Amazon Web services WSMO")
      value(ModelAuthor  "http://www.aston.ac.uk/hwang")
      value(hasSpecLanguage
         "http://www-users.aston.ac.uk/wangh10/OZ#")
   ... ... )
```

Specification model. It shows the content of a specification model and is a superclass of every software model. It is related to the class *Specification* by the property *specify* and its inverse property *specifiedBy*.

The definition of *SpecificationModel* is open to the users. Users can define their own Semantic Web environment for their own specification language. The user's model will be a subclass of the class *SpecificationModel*. Each model has a property *useLanguage* with a reference to the specification Semantic Web scheme. This schema ontology defines the Semantic Web environment for the specification language being used. Ideally there will be a standard Semantic Web environment defined for each kind of specification language. We demonstrate how the Semantic Web environment for Z/OZ can be built as follows.

Semantic Web environment for Z/OZ. The specification language providers define the Semantic Web environment for the language according to its syntax and semantics. This definition (an OWL ontology) provides information about the interpretation of the software model given in an RDF instance. For example, an OWL ontology for OZ can be developed accordingly. Part of the ontology definitions (for constructing a Z schema and OZ class) is as follows:

```
Namespace(Z=<http://www-users.aston.ac.uk/wangh10/Z#>)
Namespace(OZ=<http://www-users.aston.ac.uk/wangh10/OZ#>)
<!-- some definition omitted -->
Class(Z:Schemadef)
Class(Z:Schemabox partial Z:Schemadef
     restriction(Z:name cardinality 1)
     restriction(Z:del minCardinality 0)
     restriction(Z:decl minCardinality 0)
     restriction(Z:predicate minCardinality 0))
       ....)
Class(OZ:State partial OZ:SchemaBox
      restriction(Z:name} hasValue "")
      restriction(Z:decl cardinality 0))
Class(OZ:ClassDef partial
      restriction(Z:name cardinality 1)
```

```
restriction(OZ:superclass minCardinality 0)
restriction(OZ:stateSchema cardinality 1)
restriction(OZ:stateSchema allValueFrom OZ:State)
restriction(OZ:operation} minCardinality 0)
....)
```

The OWL class *Schemadef* represents the Z schemas. The class *Schemabox*, a subclass of *Schemadef*, represents the Z schemas defined in the schema box form. The class *Schemabox* models a type whose instance may consist of a *name*, a number of declarations *decl* and some *predicate* definitions. The class *ClassDef* represents the OZ classes whose instance must have a *name*, a *stateSchema* which is a subclass of *Schemadef* with empty name and without 'delta' declaration, and a number of *superclass* and *operations*. The following OWL definition represents the OZ class *Capability*, defined to model the functional aspects of a WSMO Web Service[3].

```
....
Individual(Capability  type(OZ:ClassDef)
     value(Z:name "Capability")
    value(OZ:superclass  ServiceElement )
     ... ... )
 ... ...
```

3.2 Ontology for the Application Domain

In the previous subsection, we proposed a framework that can be used to represent Z/OZ models in the Semantic Web environment. In order to effectively annotate formal software models, it is also important to create proper ontology for the related application domain. However, designing a clear and consistent ontology is not a trivial job. In this subsection we demonstrate the usage of TAO tools to (semi-) automatically create domain ontologies from the formal models. This can be achieved in two main steps. Firstly, the skeleton of the domain ontology is automatically extracted from formal OZ models. Secondly, by using TAO tools, the extracted ontology is further enriched and refined.

Extracting the skeleton of domain ontology from the OZ model. Ontology is an explicit specification of conceptualisation for an application domain. The formal models provide a natural basis for creating the domain ontologies. In this subsection, we demonstrate how to automatically extract domain ontologies from OZ formal models. The ontology for the system can be resolved readily from the static parts of OZ design documents. A set of transformation rules from an OZ model to its corresponding OWL ontology is developed. For example:

- The given types in the Z/OZ model are directly transformed into OWL classes.
- The transformation from functions and relations in Z/OZ, e.g. $R{:}B{\leftrightarrow}(\rightarrow,\twoheadrightarrow)C$, to OWL ontology requires several cases. The relation R is transformed into an OWL property with B as the domain class and C as the range class. For total functions

[3] http://www-users.aston.ac.uk/~wangh10/report.pdf

we restrict the $OWL : cardinality$ property to be one and for partial functions we restrict the $owl : maxCardinality$ property to be one.
- The translation of Z/OZ subset definition, e.g. $M : \mathbb{P}\, N$, depends on the translation of N. If N corresponds to an OWL class, then M is transformed into an OWL subclass of N. If N corresponds to an OWL property, then M is transformed into an OWL subproperty of N.
- For the Z/OZ constant, e.g. $x : Y$, X is transformed into an instance of Y.
- A Z/OZ state schema can be transformed into an OWL class. Its attributes are transformed into OWL properties with the schema name as the domain OWL class and the type declaration as the range OWL class.
- An OZ class can be translated into an OWL class. Its attributes defined in state schema are translated into OWL properties with the class name as its domain and the type declaration as its range. Other translation details are similar to the Z/OZ state schema translation defined above.

From the OZ class *Capability* presented earlier, the following ontology can be extracted. This extraction process can be achieved automatically.

```
<!-- some definition omitted -->
Class(Expression)
Class(ServiceElement)
Class(Capability  partial ServiceElement )
Property(hasPrecondition domain(Capability) range(Expression))
  ... ...
Individual(amazonWSCapability  type(Capability))
```

Enrich the domain ontology with TAO tools. The ontologies extracted from Z/OZ models in the previous subsection only contain high-level concepts of an application domain. In order to produce effective annotations, the extracted ontology needs need to be further enriched and refined to represent a complete knowledge about the application domain. In this subsection, we introduce the usage of *TAO tools* developed by TAO project for semi-automatic acquisition of domain ontologies, based on the state-of-the-art ontology learning and semantic data integration. We use the ontology derived from the previous subsection as the basis, and try to discover other concepts and relations using all the related system artefacts in the application domain, e.g., the source code, accompanying documentations, and external sources (such as the Web forums), based on the TAO methodology. TAO has developed various software components to support this methodology. All these software components are included in a prototype tool and is available as TAO Suite.

LATINO[4], a part of the TAO Suite, is used for Ontology Learning. LATINO is a data-mining framework that joins text mining and link analysis for the purpose of (semi-automated) ontology construction. LATINO has at least two novelties comparing with existing ontology learning methods. Firstly, using LATINO, the ontologies are constructed from the knowledge extracted from various data sources that accompany

[4] http://www.tao-project.eu/researchanddevelopment/demosanddownloads/ontology-learning-software.html

typical software applications. A set of important data sources related to the functionalities of a software application and their inter/intro-relationships are carefully studied and made use of during the ontology construction process. Secondly, LATINO is not only limited to textual data sources. Additional data resources that can be used for ontology learning include structure documents, such as database schema, UML models, existing source code, API, etc., or textual documents, such as requirement documents, manuals, forum discussions etc. For more information about the potential data sources that may be related to the description of a software systems and their classification, please refer to [1]. For the WSMO Amazon Web Service example, the ontology extracted from the OZ model is a very important learning resource for LATINO. Furthermore, the relevant Java source code, JavaDoc files and Amazon Web Service WSDL definitions are obtained for ontology learning as well. Each of the collected documents are stored in the TAO knowledge store, together with the *URLs* from which the documents were downloaded (*docURL*) and the types of the documents (*docType*), such as paper, forum post, database schema, and source code.

To use TAO method and tools for ontology learning, the first question that needs to be answered by a software engineer is – what are the *text-mining instances* (which are used as graph vertices when dealing with the structure). In this particular case, i.e., the user need to study the data at hand and decide which data entities will play the role of instances in the transitioning process. Some potential choices include Java/C++ classes, methods, database entities, etc. In our Amazon A2S example, we mainly use the existing ontology entities, java classes and XML schemas as text-mining instances.

The text-mining algorithms employed by LATINO (and also by many other data-mining tools) work with feature vectors. Therefore, once the text-mining instances have been enriched with the textual documents we need to convert them into feature vectors[5]. LATINO is able to compute the feature vectors from a document network. For the usage of LATINO, please refer to [10].

These feature vectors are further used as an input for OntoGen[6], which is a data-driven ontology construction tool that creates suggestions for new concepts for the ontology automatically. OntoGen has been integrated with TAO Suite. The most important step of ontology development is identifying the concepts in a domain. With OntoGen, this can be achieved by using either a fully automated approach such as unsupervised learning (e.g. clustering), or a semi-automated approach such as supervised learning (e.g. classification).

In the unsupervised approach, the system provides suggestions for possible sub-concepts of the selected concept. While the supervised approach is based on Support Vector Machines (SVM) active learning method, which is a set of related supervised learning methods used for classification and regression. The user can start this method by submitting a query. After the user enters a query, the active learning system starts asking questions and labelling the instances. On each step, the system asks whether a particular instance belongs to the concept. The main advantage of the unsupervised methods is that it requires very little input from the user. The unsupervised methods

[5] The feature vectors for the Amazon case study can be found at `http://www-users.aston.ac.uk/~wangh10/Amazon_DocSimRel.Bow`.

[6] http://ontogen.ijs.si/

provide well-balanced suggestions for sub-concepts based on the instances and are also good for exploring the data. The supervised method on the other hand requires more user interactions. The user has to figure out what the sub-concept is, then to describe the sub-concept trough a query and finally go through the sequence of questions to clarify the query. This is intended for the cases where the user has a clear idea of the sub-concept to be added to the ontology, while as the unsupervised methods is not capable to discover it.

For the Amazon example, we have chosen the unsupervised approach. Part of the generated concepts visualised using OntoGen is shown in Figure 3.

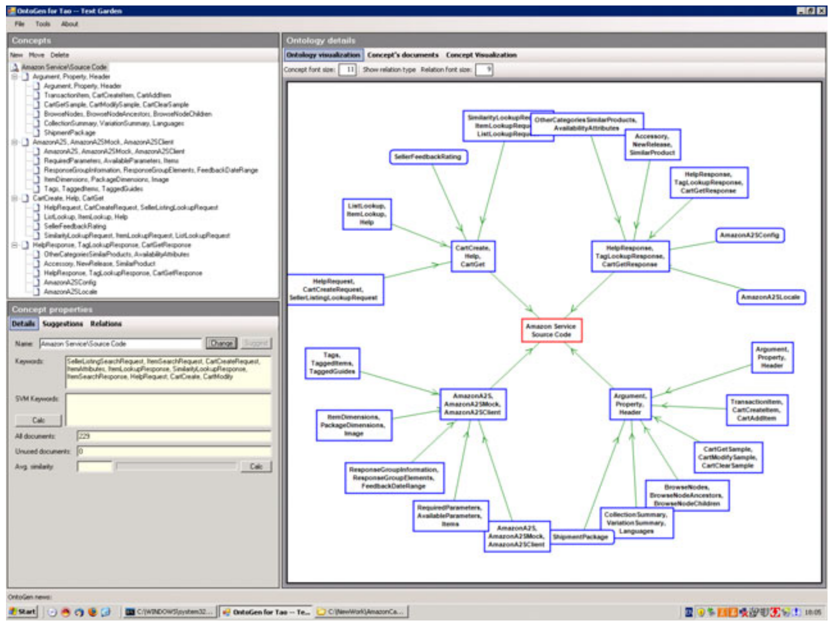


Fig. 3. Ontology concepts derived using OntoGen

After creating the domain ontology, we can save it into the TAO repository. Now we are ready to augment the content of formal models (including both models and textual explanations) and any other related software artefacts semantically.

4 Formal Model Augmentation

Formal model augmentation is a specific metadata generation task aiming at enable new information access methods. It enriches the OZ models (or other formal models) and related content with semantic information, linked to a given ontology, thus enabling semantic-based search over the annotated model contents. While there has been a significant body of research on semantic annotation of textual content (in the context of

knowledge management applications), only limited attention has been paid to process software models in particular to the problem of semantic-based software engineering. TAO has developed a tool named *Key Concept Identification Tool (KCIT)* to assist users to annotate heterogeneous software artefacts, automatically (semi-automatically). In essence, *KCIT* is capable of performing two tasks: (1) semantic annotation – using Information Extraction, some parts of the document content are marked and then linked to an ontology; (2) persistent storage and lookup of augmented content, where document retrieval is based on relevance to a selected set of semantic annotations instead of relevance to words (like ina keyword lookup). More information about *KCIT* can be found in [3].

4.1 Semantic Annotation

KCIT identifies key concepts from software model content intelligently (more than exact text match, like many other existing approaches). It can also be configured to better adopt different use cases. For example, it can be used to annotate OZ models written in LaTex format or ZML format (the XML-based ISO standard interchange format for Z/OZ models [18]). Users only need to click a button and KCIT goes through the formal models and automatically identifies the pieces of text or tag, that are related to concepts or relations defined in the domain ontology by using NLP techniques. After the process of automatic annotation is finished, users can validate the results by visualising them, correcting annotations if necessary, and adding new ones by manually selecting the text they want to link to the relevant concept from the ontology.

Figure 4 shows part of the annotation result for the WSMO Amazon Object-Z model (in LaTex) with respected descriptions. The OZ class $Capability$ (defined in LaTex as $begin\{class\}$) is associated with two annotations. Firstly, KCIT identifies it is an OZ class and annotated with the OWL individual representing the OZ class. It is defined in the ontology for representing the OZ model, as discussed in Section 3.1. Secondly KCIT recognises that this OZ class is related to the domain concept $Capability$ defined within the WSMO Amazon ontology which denotes the knowledge concept of a WSMO Web service's capability. Therefore, software engineers can easily understand what this OZ class is about and found out its functionalities. Similarly, the respected textual description is annotated. We can also use the domain ontology to annotate other related software artefacts, such as the WSMO amazon library, implementation source code that realises the OZ model, Amazon service WSDL file, etc. The output of the semantic annotation process is a set of annotations and their features – the *URL* of the ontology resource to which the term refers to, its type (e.g., an instance, a class, or a property). These semantic annotations are merged with document metadata (docURL and docType) and saved in a way that makes them accessible through semantic search.

4.2 Storing Implicit Annotations and Semantic-Based Access

In order to access the semantic knowledge, the produced annotation features, together with document-level metadata are read and exported in a format which can be easily queried via a formal language such as SPARQL. More specifically, this extracted information needs to 'connect' a document with different 'mentions' of the ontology

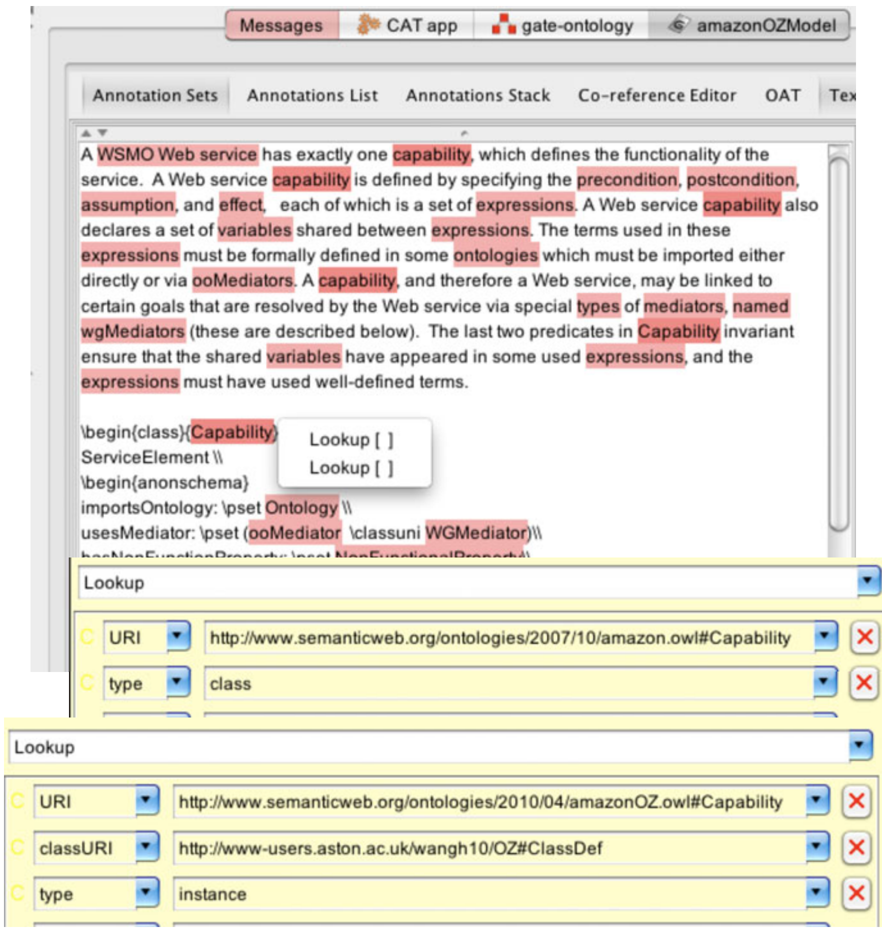


Fig. 4. Semantic Annotation over WSMO Amazon service OZ model

resources inside the documents. For example, if a model document contains mentions of the class *Capability* for the Amazon A2S, the output should be modelled in a way that preserves this information during query time (i.e., the URLs of all documents mentioning this class ca be found easily). For this purpose, the PROTON Knowledge Management ontology[7] has been used, through which the information about the type and address of a document, the position (the start and end offset) of a 'mention' within a document can be represented in a standard way.

The extracted annotations are stored in an OWL-compatible knowledge repository (OWLIM [12]), and accessible for querying using formal SW query languages (e.g. SQARQL). For example, for the OZ class *Capability*, we can easily find out where

[7] http://proton.semanticweb.org/2005/04/protonkm

this concept is explained in the manual documents, its corresponding WSDL definition and the implementation source code classes, etc. In essence, the ontology is serving as a framework that interlinks the formal models and all other software artefacts. It can be used to guide the software development process and is very helpful for software engineers to understand and maintain the software artefacts, especially for large and complex systems.

5 Conclusion

This paper proposed a framework that allows users to interconnect the knowledge about formal software models and other related documents using the semantic technology. We developed an Semantic Web environment for representing and sharing formal Z/OZ models. A method with prototype tool is presented to enhance semantic access to software models and other artefacts. This has been achieved in three steps. Firstly, the ontologies about the software model and its application domain has been automatically extracted and enriched. Secondly, the semantic annotations are automatically generated. Finally, the generated annotations together with document metadata are stored in a repository in OWL format and are made accessible to users.

By creating semantical annotations, we can transform existing software documentation into a conceptually organised and semantically interlinked knowledge space that incorporates the software models with unstructured data from multiple software artefacts: forum postings, manuals, structured data from source code and configuration files. The enriched information can then be used to add novel functionality to web-based documentation of the software concerned, providing the developer with new and powerful ways to locate and integrate components (either for reuse or for integration with new development).

Our future work will focus on the improvement of the current interface, and the implementation of query result clustering and summarisation.

References

1. Amardeilh, F., Vatant, B. Gibbins, N. Payne, T.R. Wang, H. H.: Sws bootstrapping methodology. Technical Report D1.2.2, TAO Project Deliverable (2009), `http://www.tao-project.eu/resources/publicdeliverables/d1-2-2.pdf`
2. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontology Language Reference (2004), `http://www.w3c.org/TR/owl-ref/`
3. Bontcheva, K., Damljanovic, D., Aswani, N., Agatonovic, M., Sun, J., Amardeilh, F.: Key concept identification and clustering of similar content. Technical Report D3.1, TAO Project Deliverable (2007), `http://www.gate.ac.uk/projects/tao/webpage/deliverables/d3-1.pdf`
4. Bowen, J.P., Chippington, D.: Z on the web using java. In: Bowen, J.P., Fett, A., Hinchey, M.G. (eds.) ZUM 1998. LNCS, vol. 1493, pp. 66–80. Springer, Heidelberg (1998)
5. Ciancarini, P., Mascolo, C., Vitali, F.: Visualizing z notation in html documents. In: Bowen, J.P., Fett, A., Hinchey, M.G. (eds.) ZUM 1998. LNCS, vol. 1493, pp. 81–95. Springer, Heidelberg (1998)

6. D. Brickley and R.V. Guha (eds.): Resource description framework (rdf) schema specification 1.0 (February 2004), http://www.w3.org/TR/rdf-schema/
7. Dong, J.S., Sun, J., Wang, H.: Semantic web for extending and linking formalisms. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 587–606. Springer, Heidelberg (2002)
8. Duke, R., Rose, G.: Formal Object Oriented Specification Using Object-Z. In: Cornerstones of Computing. Macmillan, Basingstoke (March 2000)
9. Futatsugi, K., Nakagawa, A.: An overview of cafe specification environment - an algebraic approach for creating, verifying, and maintaining formal specifications over networks. In: ICFEM 1997: Proceedings of the 1st International Conference on Formal Engineering Methods, Washington, DC, USA, p. 170. IEEE Computer Society, Los Alamitos (1997)
10. Grcar, M.: Ontology learning services library. Technical Report D2.2.2, TAO Project Deliverable (2008), http://www.gate.ac.uk/projects/tao/webpage/deliverables/d2-2-2.pdf
11. Jacek, K., Dumitru, R., James, S.: Wsmo use case: Amazon e-commerce service (2006) (unpublished manuscript)
12. Kiryakov, A., Ognyanov, D., Manov, D.: Owlim a pragmatic semantic repository for owl. In: Int. Workshop on Scalable Semantic Web Knowledge Base Systems, New York City, USA, pp. 182–192. Springer, Heidelberg (2005)
13. Lassila, O., Swick, R.R. (eds.): Resource description framework (rdf) model and syntax specification (February 1999), http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
14. Lee, B.T., Hendler, J., Lassila, O.: The semantic web. Scientific American (May 2001)
15. Smith, G.: The Object-Z Specification Language. In: Advances in Formal Methods. Kluwer Academic Publishers, Dordrecht (2000)
16. Spivey, J.M.: The Z Notation: A Reference Manual. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1989)
17. Sun, J., Dong, J.S., Liu, J., Wang, H.: Object-Z Web Environment and Projections to UML. In: WWW-10: 10th International World Wide Web Conference, pp. 725–734. ACM Press, New York (May 2001)
18. Sun, J., Dong, J.S., Liu, J., Wang, H.: A formal object approach to the design of zml. Annals of Software Engineering, an International Journal 13, 329–356 (2002)
19. van Harmelen, F., Patel-Schneider, P.F., Horrocks, I. (eds.): Reference description of the daml+oil ontology markup language. Contributors: T. Berners-Lee, D. Brickley, D. Connolly, M. Dean, S. Decker, P. Hayes, J. Heflin, J. Hendler, O. Lassila, D. McGuinness, L. A. Stein... (March 2001)
20. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall, Englewood Cliffs (1996)

Making Pattern- and Model-Based Software Development More Rigorous

Denis Hatebur[1,2] and Maritta Heisel[1]

[1] Universität Duisburg-Essen, Germany, Fakultät für Ingenieurwissenschaften
maritta.heisel@uni-due.de
[2] Institut für technische Systeme GmbH, Germany
d.hatebur@itesys.de

Abstract. Pattern-based and model-based software development approaches have a high potential to improve the quality of software. Patterns allow engineers to re-use established and proven development knowledge. Developing software by constructing a sequence of models provides engineers with various possibilities for validation, because the different development models are not independent of each other and hence can be checked for coherence.

We present a UML profile equipped with numerous OCL constraints that supports a pattern- and model-based software development process. The basis of the UML profile is a representation of problem frames, which are patterns supporting requirements analysis. OCL constraints provide a formal underpinning of the development process and allow one to perform semantic checks every time a new model is set up. Our approach is supported by a tool, called UML4PF. The tool is based on the Eclipse development environment, extended by an EMF-based UML tool, in our case, Papyrus. In this paper, we specifically focus on ensuring that problem frames are instantiated correctly. We illustrate our approach by the case study of an automatic teller machine.

1 Introduction

Software development with formal methods usually starts with a formal specification. Then, this specification is refined to code, possibly carrying out several refinement steps that must be proven correct. Another possibility is to annotate code with formally expressed assertions and prove the correctness of the code with respect to the assertions.

Today, patterns do not play a prominent role in such formal development processes. However, patterns are a very important means to reuse development knowledge. Therefore, they should also be integrated in formal development processes. Patterns are defined for all stages of the software development process. Requirements analysis can be supported by problem frames [16] and analysis patterns [9]. Coarse-grained design can make use of architectural styles [21]. Design patterns [10] are a well-known means to perform fine-grained design. Idioms [5] support programming, and test patterns [18] can be used in the testing phase.

Since the different artifacts of the software development process can be expressed as models, pattern- and model-based development fit very well together. The advantage of model-based development is that it is possible to check integrity conditions between the

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 253–269, 2010.
© Springer-Verlag Berlin Heidelberg 2010

models. When such a check is performed by formal means, pattern- and model-based development processes can be carried out in a formal way.

In this paper, we present a pattern- and model-based requirements analysis process that is tool-supported and that allows one to check semantic integrity conditions that are expressed in the formal language OCL[1] (Object Constraint Language) [23]. We thus contribute to the goal of combining the use of patterns with formal development. That process is based on problem frames [16]. These are patterns that classify software development problems.

We have defined a UML profile that extends the UML meta-model [25]. It allows us to express the diagrams that are set up when performing requirements analysis with problem frames in UML notation. The defined OCL conditions provide a formal semantic underpinning of the problem frame approach. Automatically checking the constraints makes it possible to detect semantic errors in the requirements analysis process.

We have developed a tool, called UML4PF. With this tool, developers can draw different diagrams that have to be set up during the requirements engineering process. The diagrams are mapped to parts of a global model, and a graphical representation of this part. Every time a new diagram is finished, the developer may call the UML4PF validator. This causes the defined OCL conditions to be evaluated, based on the model information. If one of the conditions is not satisfied, a semantic error has been detected in one of the diagrams, or integrity conditions between two or more diagrams are violated. UML4PF also points out which condition is violated in which diagram(s), thus supporting the developer in locating and correcting the error.

Elements of the created model can be re-used in later development phases. We can also validate that the artifacts of later development steps, such as specification and architectural design, are consistent with the requirements engineering diagrams.

In the following, we introduce problem frames and the corresponding UML profile in Section 2. The tool UML4PF is described in Section 3. In Section 4, we show how to check that problem frames are correctly instantiated during a development[2]. Section 5 illustrates the approach by the case study of an automatic teller machine. Section 6 discusses related work. Finally, Sect. 7 concludes the paper with a summary, ongoing work, and directions for future research.

2 UML Profile for Problem Frames

Problem frames are a means to describe software development problems. They were introduced by Jackson [16], who describes them as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."*

Figure 1 shows a problem frame called *commanded behaviour* in UML notation. Informally, *there is some part of the physical world whose behaviour is to be controlled with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.* [16]. We describe

[1] We have chosen OCL because it is part of UML, which is widely used and well equipped with tool support.

[2] Many other checks are defined that are not presented in this paper.

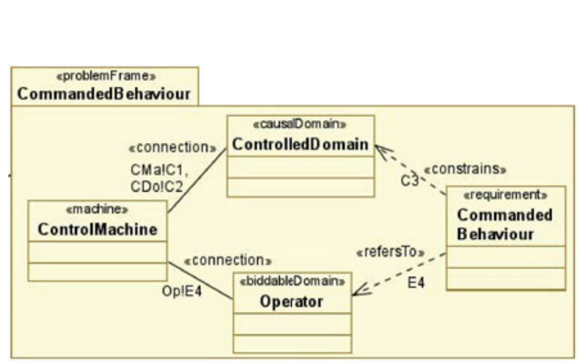


Fig. 1. *Commanded behaviour* problem frame using UML notation

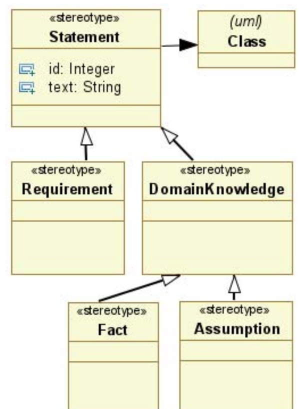


Fig. 2. Requirement stereotype inheritance structure

problem frames using class diagrams extended by stereotypes (see Fig. 1). All elements of a problem frame diagram act as placeholders, which must be instantiated to represent concrete problems. Doing so, one obtains a problem description that belongs to a specific problem class.

The class with the stereotype <<*machine*>> represents the software to be developed (possibly complemented by some hardware). The classes with domain stereotypes (e.g., <<*CausalDomain*>> or <<*BiddableDomain*>>) represent *problem domains* that already exist in the application environment.

In frame diagrams, *interfaces* connect domains, and they contain *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Fig. 1 the notation Op!E4 means that the phenomena in the set E4 are controlled by the domain Operator. These interfaces are represented as associations, and the name of the associations contain the phenomena and the domain controlling the phenomena.

The associations can be replaced by interface classes, whose operations correspond to phenomena. The interface classes are either controlled or observed by the connected domains, represented by dependencies with the stereotypes <<*controls*>> or <<*observes*>>. Each interface can be controlled by at most one domain. A controlled interface must be observed by at least one domain, and an observed interface must be controlled by exactly one domain.

Problem frames substantially support developers in analyzing problems to be solved. They show what domains have to be considered, and what knowledge must be described and reasoned about when analyzing the problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the *domain knowledge*.

The domain knowledge consists of *assumptions* and *facts*. Assumptions are conditions that are needed, so that the *requirements* are accomplishable. Usually, they

describe required user behavior. For example, it must be assumed that a user ensures not to be observed by a malicious user when entering a password. Facts describe fixed properties of the problem environment, regardless of how the machine is built.

Domain knowledge and requirements are special statements. A statement is modeled similarly to a SysML requirement [24] as a class with a stereotype. In this stereotype a unique identifier and the statement text are contained as stereotype attributes. Fig. 2 shows the stereotype Statement that extends the metaclass Class of the UML metamodel.

When we state a requirement, we want to change something in the world with the machine to be developed. Therefore, each requirement constrains at least one domain. This is expressed by a dependency from the requirement to a domain with the stereotype <<*constrains*>>. Such a constrained domain is the core of any problem description, because it has to be controlled according to the requirements. Hence, a constrained domain triggers the need for developing a new software (the machine), which provides the desired control.

A requirement may refer to several domains in the environment of the machine. This is expressed by a dependency from the requirement to a domain with the stereotype <<*refersTo*>>. The referred domains are also given in the requirements description.

In Fig. 1, the Controlled Domain domain is constrained, because the Control Machine has the role to change it on behalf of user commands for achieving the required Commanded Behaviour.

Jackson distinguishes the domain types *biddable domains* that are usually people, *causal domains* that comply with some physical laws, and *lexical domains* that are data representations. The domain types are modeled by the stereotypes <<*BiddableDomain*>> and <<*CausalDomain*>> being subclasses of the stereotype <<*Domain*>>. A lexical domain (<<*LexicalDomain*>>) is modeled as a special case of a causal domain. To describe the problem context, a connection domain between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. They are modeled as classes with the stereotype <<*ConnectionDomain*>>. Connection domains are, e.g., video cameras, sensors, or networks. This kind of modeling allows one to add further domain types, such as <<*DisplayDomain*>> (introduced in [6]) being a special case of a causal domain. Figure 3 depicts the domain stereotypes defined in our UML Profile.

Other problem frames besides the commanded behavior frame are *required behaviour*, *simple workpieces*, *information display*, and *transformation* [16].

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements (see Fig. 5 for an example). Then, the problem is decomposed into subproblems. If possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame, one must instantiate its frame diagram, i.e., provide instances for its domains, phenomena, and interfaces. The instantiated frame diagram is called a *problem diagram*.

The different diagram types make use of the same basic notational elements. As a result, it is necessary to explicitly state the type of diagram by appropriate stereotypes.

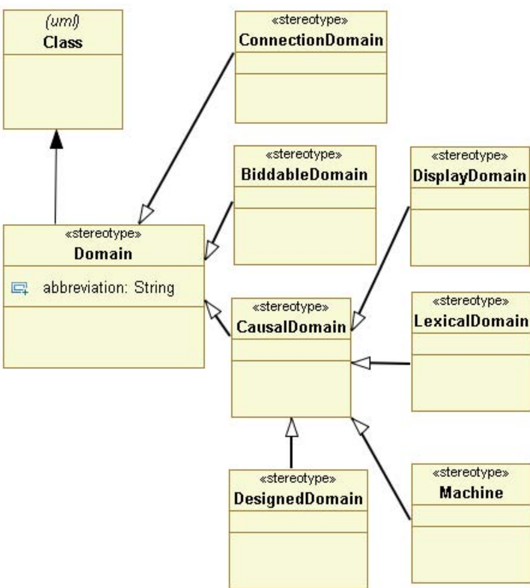


Fig. 3. Domain stereotypes in UML Profile

In our case, the stereotypes are <<*ContextDiagram*>>, <<*ProblemDiagram*>>, and <<*ProblemFrame*>>. These stereotypes extend (some of them indirectly) the metaclass Package in the UML meta-model.

Successfully fitting a problem to a given problem frame means that the concrete problem indeed exhibits the properties that are characteristic for the problem class defined by the problem frame. A problem can only be fitted to a problem frame if the involved problem domains belong to the domain types specified in the frame diagram. For example, the Operator domain of Fig. 1 can only be instantiated by persons, but not for example by some physical equipment like an elevator. Thus, an advantage of using problem frames in requirements engineering is that problems are mapped to well-known problem classes that are practically relevant. Moreover, when using problem frames, one can even hope for more than just a full comprehension of the problem at hand. Since problems fitting to a problem frame share common properties, their solutions will share common properties, too [2]. Thus, problem frames provide pattern-based support not only for problem comprehension, but also for problem solving. For each subproblem, a separate architecture can be developed as described in [2]. These can be merged in a systematic way, see [3].

3 Tool Support

We have developed a tool called UML4PF to support the requirements engineering process sketched in Section 2 as well as subsequent development steps, such as deriving software architectures from problem descriptions. After the developer has drawn some

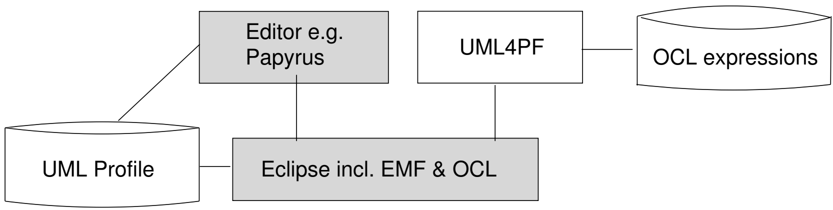


Fig. 4. Tool Realization Overview

diagram(s) using some EMF-based editor, for example Papyrus UML [20], UML4PF provides him or her with the following functionality:

- It checks if the developed model is valid and consistent by using our OCL constraints.
- It returns the location of invalid parts of the model.
- It automatically generates model elements, e.g., it generates observed and controlled interfaces from association names.

Figure 4 provides an overview of the context of our tool. Gray boxes denote re-used components, whereas white boxes describe those components that we created. Basis is the Eclipse platform [7] together with its plug-ins EMF [8] and OCL [23]. These plug-ins provide functions to query a model with OCL. Our UML-profile is conceived as an Eclipse plug-in, extending the EMF meta-model. We store the data in the profile in XMI-format. We store all our OCL constraints in one file in XML-format. They are directly checked using the OCL executor, which is part of EMF.

The graphical representation of the different diagram types can be manipulated by using any EMF-based editor. We selected Papyrus UML [20] as it is available as an Eclipse plug-in, open-source, and EMF-based. UML4PF provides additional windows in Eclipse to edit requirements and traceability links as an easy-to-use user interface. The requirements and traceability links are directly stored in the UML model. The graphical representation of the created UML elements is not necessary, but can be added later. The tool is an open source tool under development and is available on demand from the authors.

Listing 1 shows an example of an integrity condition. It formalizes the general fact that each statement constrains (see Fig. 2) constrains at least one domain. All classes in the model (Line 1) with the stereotype <<*Statement*>> (accessed by the EMF keyword getAppliedStereotypes) or a specialized statement subtype[3], e.g., <<*Requirement*>>) (Lines 2-5) are selected. For these classes, the dependencies of class (clientDependency) (Line 6) with the stereotype <<*constrains*>> are collected (Line 7). The number of 'constrains' for each class must be bigger than or equal to one (Line 8). In another OCL integrity condition, it is stated that all <<*constrains*>> dependencies must point to domains.

[3] The superclass can be accessed by the EMF keyword general. Since the keyword is not recursive, we need to address each of the 3 possible hierarchy levels explicitly.

```
1 Class.allInstances()->select(
2 getAppliedStereotypes().name->includes('Statement') or
3 getAppliedStereotypes().general.name ->
      includes('Statement') or
4 getAppliedStereotypes().general.general.name ->
      includes('Statement') or
5 getAppliedStereotypes().general.general.general.name ->
      includes('Statement'))
6 ->forAll(clientDependency->collect(d |
7     d.oclAsType(Dependency).getAppliedStereotypes().name ->
          includes('constrains '))
8     ->count(true)>=1)
```

Listing 1. Statements have at least one Constrains Dependency

4 Checking the Correct Instantiation of Problem Frames

This section, we present a number of OCL constraints that can be used to check if a given problem diagram is a correct instantiation of a given problem frame. Such checks are very important, because a software development problem only belongs to the problem class characterized by the problem frame if it really exhibits all characteristics required by the frame. Only then can the solution approaches associated with the problem frame by successfully applied.

We have also defined other OCL constraints (not presented in this paper) that concern the relation between context diagrams and problem diagrams as well as the consistency between problem diagrams and behavioral descriptions, expressed as sequence diagrams. Another paper [13] presents constraints for describing dependability requirements, such as confidentiality, integrity, and reliability. In this paper, we specifically focus on ensuring that problem frames are instantiated correctly.

For each problem diagram, we explicitly state which problem frame it instantiates by using a dependency with the stereotype <<*instanceOf*>>. The OCL expression of Listing 2 checks if the stereotype <<*instanceOf*>> is used correctly. To this end, all dependencies in the model (Line 1) with the stereotype <<*instanceOf*>> (accessed by the EMF keyword getAppliedStereotypes) (Line 2) are selected. For these dependencies (Line 3), the source and the target must be a package (checked by the EMF expression oclIsTypeOf(Package) (Lines 4 and 5), the source package has the stereotype <<*ProblemDiagram*>> (Line 6), and the target package has the stereotype <<*ProblemFrame*>> (Line 7).

```
1 Dependency.allInstances()
2 ->select(a |
      a.oclAsType(Dependency).getAppliedStereotypes().name ->
      includes('instanceOf ') )
3 ->forAll(d |
4    d.oclAsType(Dependency).source->forAll(oclIsTypeOf(Package))
        and
```

```
5   d.oclAsType(Dependency).target->forAll(oclIsTypeOf(Package))
      and
6   d.oclAsType(Dependency).source.getAppliedStereotypes().name
      -> includes('ProblemDiagram') and
7   d.oclAsType(Dependency).target.getAppliedStereotypes().name
      -> includes('ProblemFrame'))
```

Listing 2. 'instanceOf'-Dependencies are from ProblemDiagrams to ProblemFrames

If a problem diagram correctly instantiates a problem frame, possible solutions defined for the problem frame can be reused for the concrete problem. For example, corresponding architectural patterns [2] can be applied.

For security-related problems (see, e.g., [14]), we are not allowed to add additional interfaces, whereas for other software development problems, additional elements are allowed to be added to the problem diagram. Therefore, we distinguish between two kinds of instances, namely *strict* and *weak* instances. In the OCL expressions, we only use the predicate weak.

We now present a set of conditions that should evaluate to true if a given problem diagram is a valid instantiation of a given problem frame. These OCL constraints are one of the contributions of this paper. Some of them we have derived from the informal explanations given by Jackson [16], for example conditions 1, 3, 5 and 7 given below. With these conditions (together with the rules given in [15]), we provide the problem frame approach with a formal semantic underpinning. Other conditions express general rules about correctly instantiating patterns, e.g., conditions 2, 3[4] and 6. All conditions are decidable, because they check semantic properties of problem descriptions that are expressed as syntactic properties of the corresponding UML models.

Our UML profile is not an exact match of the problem frame approach, but provides several enhancements. One of them is the distinction between weak and strict instantiations. Condition 5 states how a weak instance is distinguished from a strict one.

We do not claim that the integrity conditions we have defined so far are complete. On the contrary, it is easily possible to identify new conditions and incorporate them into UML4PF. In any case, it is impossible to come up with a set of semantic integrity condtions that is *sufficient* for the correctness of the defined models. However, the conditions constitute *necessary* conditions for the correctness of the defined models. Therefore, a violation of one of the conditions really indicates an error in the development.

For a given problem diagram to be a valid instantiation of a given problem frame, the following conditions should evaluate to true:

1. The domain types of the constrained domains in the problem frame are the same as in the problem diagram.
2. Each domain referred to by the requirement in the problem frame corresponds to a domain in the problem diagram (same domain types).
3. Each connection in the problem frame corresponds to a connection in the problem diagram, i.e., they connect same domain types.

[4] This condition combines a general instantiation rule with a problem-frame-specific rule.

4. For strict (i.e., non-weak) instances, each connection in the problem diagram corresponds to a connection in the problem frame, i.e., they connect the same domain types.
5. The domain types in problem diagrams and problem frames are consistent: the number of domains of each type in the problem frame is equal to the number of this type in the problem diagram. In case of a weak instance, the number of domains of each type in the problem frame is smaller than or equal to the number of this type in the problem diagram.
6. For strict instances, the direction of the interfaces (observed vs. controlled) is the same in the problem diagram and the problem frame. We allow that interfaces are left out.
7. Interfaces cannot be left out if they are controlled by the machine.

In the following, we present the OCL expressions checking a selection of these conditions.

Condition 1. In the OCL expression of Listing 3, all dependencies in the model (Line 1) with the stereotype <<*instanceOf*>> (Line 2) are selected. For these dependencies (Line 3) the parts of the source (the problem diagram) being requirements (Lines 4 and 5) are selected. For these requirements, the dependencies with the stereotype <<*constrains*>> are selected (Line 6). The target of these dependencies are the constrained classes, and the bag of their stereotype names (Line 7) must be the same (Line 8) as the bag of stereotype names of constrained domains in the problem frame (Lines 9-13).

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf'))
2  ->forAll(
3    source.oclAsType(Package)
4    .clientDependency -> select(getAppliedStereotypes().name ->
         includes('isPart'))
5    .target -> select(getAppliedStereotypes().name ->
         includes('Requirement')).oclAsType(Class)
6    .clientDependency -> select(getAppliedStereotypes().name ->
         includes('constrains')).
7    target.getAppliedStereotypes().name
8    =
9    target.oclAsType(Package)
10   .clientDependency -> select(getAppliedStereotypes().name ->
         includes('isPart'))
11   .target -> select(getAppliedStereotypes().name ->
         includes('Requirement')).oclAsType(Class)
12   .clientDependency -> select(getAppliedStereotypes().name ->
         includes('constrains')).
13   target.getAppliedStereotypes().name
14 )
```

Listing 3. Constrained domain type in ProblemDiagrams and ProblemFrames

Condition 2 can be checked in a similar way.

Condition 3. Line 1 in the OCL expression of Listing 4 selects all dependencies with the stereotype <<*instanceOf*>>. For all such dependencies (Line 2), we select all associations (Line 3) connecting classes (Line 4) and all association whose ends (Lines 5 and 6) are part of the problem frame the <<*instanceOf*>>-dependency points to (Lines 7-12). For all such associations in the problem frame (Line 13), we select all associations (Line 14) connecting classes (Line 15) and all associations whose ends (Lines 16 and 17) are part of the problem diagram the <<*instanceOf*>>-dependency comes from (Lines 18-23). We verify in Line 24 that in the selected set of problem diagram associations, an association exists that connects classes with the same stereotypes (Line 25 and 26).

```
 1 Dependency.allInstances() ->
       select(getAppliedStereotypes().name ->
       includes('instanceOf'))
 2 ->forAll(inst_of_dep |
 3    Association.allInstances()
 4    ->select(endType ->forAll(oclIsTypeOf(Class)))
         .oclAsType(Association)
 5    ->select(ass |
 6       ass.oclAsType(Association).endType->forAll (ass_end |
 7          inst_of_dep.oclAsType(Dependency)
 8          .target.oclAsType(Package)
 9          .clientDependency ->
                select(getAppliedStereotypes().name ->
                includes('isPart'))
10          .target -> select(oclIsTypeOf(Class)).oclAsType(Class)
               -> asSet()
11         -> includes(ass_end.oclAsType(Class))
12       )
13    )->forAll(ass_in_pf |
14       Association.allInstances()
15       ->select(endType ->forAll(oclIsTypeOf(Class)))
            .oclAsType(Association)
16       ->select(ass |
17          ass.oclAsType(Association).endType->forAll (ass_end |
18       inst_of_dep.oclAsType(Dependency)
19       .source.oclAsType(Package)
20       .clientDependency -> select(getAppliedStereotypes().name
            -> includes('isPart'))
21       .target->select(oclIsTypeOf(Class)).oclAsType(Class) ->
            asSet()
22       ->includes(ass_end.oclAsType(Class))
23          )
24       )->exists(ass_in_pd |
25          ass_in_pd.endType.oclAsType(Class)
               .getAppliedStereotypes().name
```

```
26        -> includesAll (ass_in_pf.endType.oclAsType(Class)
               .getAppliedStereotypes().name)
27      )
28    )
29  )
```

Listing 4. Connections in ProblemDiagrams and ProblemFrames are consistent

Condition 4 can be checked in a similar way.

Condition 5. Line 1 in the OCL expression in Listing 5 selects all dependencies with the stereotype <<*instanceOf*>>. For these dependencies (Line 2), we define the boolean variable weak as the value of the attribute weak of the dependency (Lines 3-5), we define the bag pf_domain as all domains of the problem frame the dependency points to (Lines 6-10), and we define the bag pd_domains as all domains of the problem diagram (Lines 11-15).

```
1  Dependency.allInstances() ->
       select(getAppliedStereotypes().name->includes('instanceOf')
       )
2  ->forAll( inst_of_dep |
3    let weak: Boolean =
4      inst_of_dep.getValue(inst_of_dep.oclAsType(Dependency)
           .getAppliedStereotypes()
          ->select(name->includes('instanceOf')) ->asSequence()
          ->first(),'weak').oclAsType(Boolean)
5    in
6    let pf_domains: Bag(Class) =
7      inst_of_dep.target.oclAsType(Package)
8      .clientDependency ->
           select(getAppliedStereotypes().name->includes('isPart'))
9      .target -> select(oclIsTypeOf(Class))
          ->reject(getAppliedStereotypes().name
          ->includes('Requirement')).oclAsType(Class)
10   in
11   let pd_domains: Bag(Class) =
12     inst_of_dep.source.oclAsType(Package)
13     .clientDependency ->
           select(getAppliedStereotypes().name->includes('isPart'))
14     .target -> select(oclIsTypeOf(Class))
          ->reject(getAppliedStereotypes().name
          ->includes('Requirement')).oclAsType(Class)
15   in
16     pf_domains->forAll(domains |
17     domains.getAppliedStereotypes().name ->forAll(stn |
18       (pf_domains->select(getAppliedStereotypes().name
             ->includes(stn))->size() =
19       pd_domains->select(getAppliedStereotypes().name
             ->includes(stn))->size())
```

```
20         or  (weak  and
21           (pf_domains->select(getAppliedStereotypes().name
                 ->includes(stn))->size()  <=
22           pd_domains->select(getAppliedStereotypes().name
                 ->includes(stn))->size())
23         )
24       )
25       )
```

Listing 5. Domain Types in Problem Diagrams and Problem Frames are Consistent

Using these definitions, we validate for all stereotype names of all problem frame domains (stn, Lines 16-17) that the number of problem frame domains with the stereotype stn is equal to the number of problem diagrams domains with this stereotype name (Lines 18-19), or for weak dependencies (Line 20) that the number of problem frame domains with the stereotype stn is smaller than or equal to the number of problem diagrams domains with this stereotype name (Lines 21-22).

5 Case Study

In this section, demonstrate how checking our OCL constraints helps developers in detecting and eliminating errors in the models they develop. As an example, we use a simplified automatic teller machine (ATM).

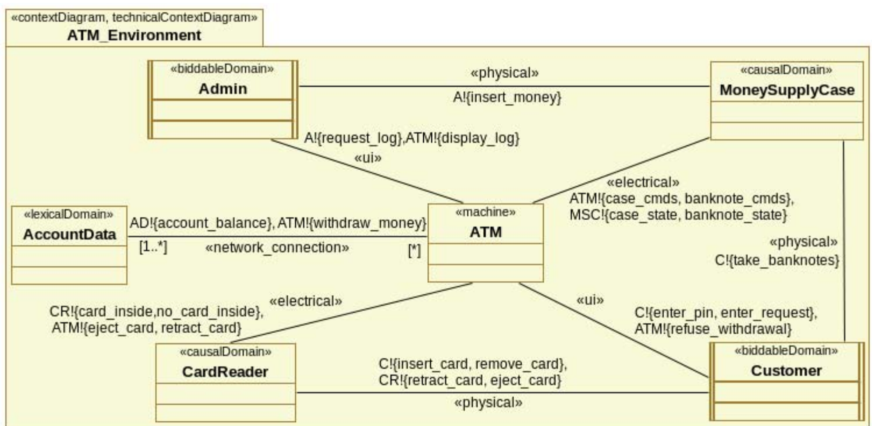


Fig. 5. ATM Context Diagram

The intended environment of the ATM is described using a context diagram as depicted in Fig. 5. It contains the ATM as the machine to be built. In the environment, we can find the Admin responsible for checking the logs of the ATM with the phenomenon request_log and for filling the MoneySupply_Case with money (phenomenon insert_money). The Customers can

– withdraw money by inserting their banking card (insert_card) into the CardReader,
– enter their PIN (enter_PIN),
– request a certain amount of money (enter_request),
– remove their card from the CardReader, and
– take money from the MoneySupply_Case.

In some cases, it is possible that the ATM refuses a withdrawal (refuse_withdrawal). Each ATM is connected with the AccountData of at least one bank. We use multiplicities to express this aspect. The different domains are annotated with appropriate stereotypes from the <<*domain*>> stereotype, e.g., the Customer is biddable and the AccountData is lexical. The connections are marked with specializations of the stereotype <<*connection*>>, e.g., a user interface (<<*ui*>>) between Customer and ATM, and a physical connection (<<*physical*>>) between Customer and CardReader.

We consider one subproblem, treating the requirement: *The money supply case supplies the banknotes as requested and retracts the banknotes if the customer does not*

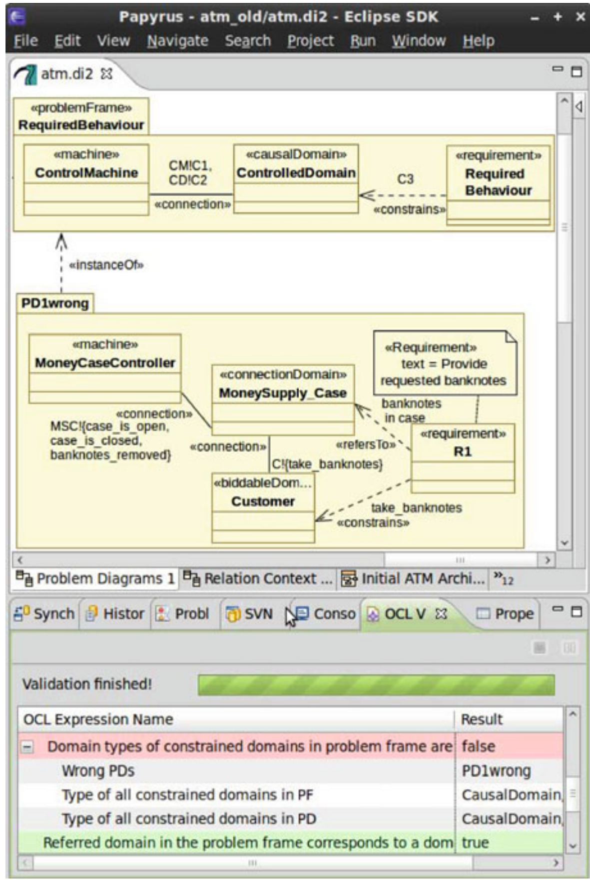


Fig. 6. Erroneous ATM Problem Diagram

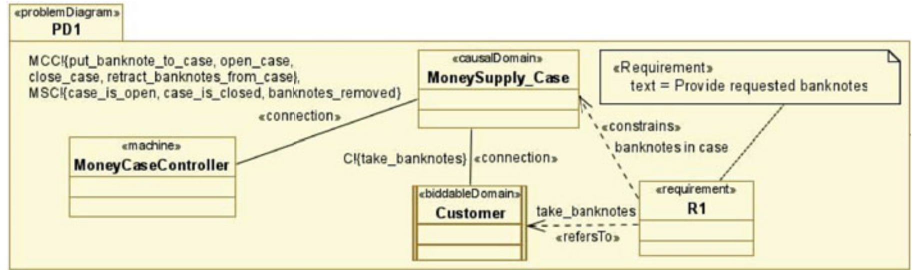


Fig. 7. ATM Control Card Reader Problem Diagram

take them. An initial problem diagram for this requirement is given in Fig. 6. It is stated to be a (strict) instance of the *required behaviour* problem frame.

Checking the OCL constraints given in Section 4 on our ATM model shows that the validation constraint given in Listing 2 is true and that our model also satisfies Condition 2. Conditions 4, 5, and 6 are not satisfied, because the additional domain Customer with its interfaces is introduced. The problem depicted in Fig. 6 is not a security problem, and the Customer should be just included in the problem diagram to describe the relevant context completely. Therefore, we decide to state that the instance is only weak. After this modification, conditions 1, 3, and 7 are still not satisfied. Condition 1 is not true, because the problem diagram contains no constrained class with the stereotype <<*CausalDomain*>>. Condition 3 is not true, because the problem frame connects the machine with a causal domain, whereas in the problem diagram, the machine is connected to a connection domain. To fulfill Conditions 1 and 3, we replace the stereotype <<*ConnectionDomain*>> with the stereotype <<*CausalDomain*>>. Condition 7 is not fulfilled, because the problem diagram contains no interfaces controlled by the machine. To solve this problem we add MCC!{put_banknote_to_case, open_case, close_case, retract_banknotes_from_case} to the machine interface. The corrected problem diagram is depicted in Fig. 7.

The complete case study consists of 4 problem diagram being instances of problem frames, 12 different domains and 33 associations. More than 50 OCL constraints were checked using our tool, which takes about 30 seconds on a standard computer. As a final result, the ATM model has been successfully validated. Figure 6 shows a screen-shot of a view on the ATM model in Eclipse with our plug-in.

6 Related Work

Lencastre et al. [17] define a meta-model for problem frames using UML. Their meta-model considers Jackson's whole software development approach based on context diagrams, problem frames, and problem decomposition. In contrast to our meta-model, it only consists of a UML class model without OCL integrity constraints. Moreover, their approach does not qualify for a meta-model in terms of MDA because, e.g., the class Domain has subclasses Biddable and Given, but an object cannot belong to two classes at the same time (c.f. Figs. 5 and 11 in [17]).

Hall et al. [12] provide a formal semantics for the problem frame approach. They introduce a formal specification language to describe problem frames and problem diagrams. However, their approach does not consider integrity conditions.

Seater et al. [19] present a meta-model for problem frame instances. In addition to the diagram elements formalized in our meta-model, they formalize requirements and specifications. Consequently, their integrity conditions ("wellformedness predicate") focus on correctly deriving specifications from requirements. In contrast, our meta-model concentrates on the structure of problem frames and the different domain and phenomena types.

We agree with Haley [11] on adding cardinality to standard problem frames to enhance the detailing of shared phenomena at the interfaces. In contrast to Haley though, we do not extend the problem frames notation by introducing a new notational element. We adopt the means provided by UML to annotate problem frames in our meta-model instead.

Van Lamsweerde [27] considers the relationships between problem worlds and machine solutions. He makes a distinction between different statement subtypes. In our profile we cover a subset of these statements. Furthermore, he introduces *Satisfaction Arguments*.

Charfi et al. [1] use a modeling framework called *Gaspard2* to design high-performance embedded systems-on-chip. They use model transformations to move from one level of abstraction to the next. To validate that their transformations have been correctly performed, they use the OCL language to specify the properties that must be checked in order to be considered as correct with respect to Gaspard2. We have been inspired by this approach. However, we do not focus on high-performance embedded systems-on-chip. Instead, we target general software development challenges.

Colombo et al. [4] model problem frames and problem diagrams with SysML [22]. They state that "*UML is too oriented to software design; it does not support a seamless representation of characteristics of the real world like time, phenomena sharing [...]*". We do not agree with this statement. So far, we have been able to model all necessary means of the requirements engineering process using UML.

Other important UML profiles are SysML [22] for system engineering and MARTE [26] for model-driven development of Real Time and Embedded Systems (RTES). The UML profile for MARTE (in short MARTE) provides support for specification, design, and verification/validation stages.

7 Conclusions and Future Work

We have shown how a pattern- and model-based requirements engineering process can be equipped with formal elements that allow developers to detect and correct errors in their models. We achieved this by means of a UML profile that allows one to express the different models being developed during the process in UML. The patterns (in particular, problem frames) can also be expressed with the profile. In this way, one can state conditions that check if a problem frame has been instantiated correctly. Besides these conditions, many other integrity conditions can be expressed in OCL. The approach is tool-supported, which is needed for its practical applicability. In particular, our contributions include the following:

- We have shown how formal and pattern-based software development can be combined.
- We provide a formal underpinning of the problem frame approach.
- We provide tool support for the problem frame approach. This tool support concerns the detection of semantic errors in the requirements engineering process.
- With the defined UML profile, we have provided a basis for a seamless model-and pattern-based development process that covers not only requirements analysis, but also specification and architectural design.
- The defined UML profile can easily be extended to cover not only several phases of software development, but also the specific treatment of dependability requirements [13] and other quality requirements.

Currently, we are augmenting our process to cover also the design phase of the software development process. We have already augmented our profile by architectural elements, and we have defined a number of OCL constraints checking the coherence of problem descriptions (i.e., context and problem diagrams) and architectural diagrams. Moreover, we have taken first steps to support software evolution. In particular, we are introducing traceability links to trace requirements to artifacts developed later, e.g. components in the software architecture.

In summary, our approach has the potential to make software development more rigourous and less error-prone, because semantic integrity conditions can be checked as soon as a new model is set up. Moreover, the use of patterns can be integrated in a natural way.

In the future, we plan to extend our tool to support the identification of missing and interacting requirements. In the long run, we aim to cover all phases of the software development process.

References

1. Charfi, A., Gamatié, A., Honoré, A., Dekeyser, J.-L., Abid, M.: Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. In: 4èmes Jounées sur l'Ingénierie Dirigée par les Modèles (IDM 2008) (2008)
2. Choppy, C., Hatebur, D., Heisel, M.: Architectural patterns for problem frames. IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures 152(4), 198–208 (2005)
3. Choppy, C., Hatebur, D., Heisel, M.: Component composition through architectural patterns for problem frames. In: Proc. XIII Asia Pacific Software Engineering Conference (APSEC), pp. 27–34. IEEE, Los Alamitos (2006)
4. Colombo, P., del Bianco, V., Lavazza, L.: Towards the integration of SysML and problem frames. In: IWAAPF 2008: Proceedings of the 3rd international workshop on Applications and advances of problem frames, pp. 1–8. ACM, New York (2008)
5. Coplien, J.O.: Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading (1992)
6. Côté, I., Hatebur, D., Heisel, M., Schmidt, H., Wentzlaff, I.: A systematic account of problem frames. In: Proceedings of the European Conference on Pattern Languages of Programs (EuroPLoP 2007), Universitätsverlag Konstanz (2008)
7. Eclipse Foundation. Eclipse - An Open Development Platform (May 2008), `http://www.eclipse.org/`

8. Eclipse Foundation. Eclipse Modeling Framework Project (EMF) (May 2008), http://www.eclipse.org/modeling/emf/
9. Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley, Reading (1997)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison Wesley, Reading (1995)
11. Haley, C.B.: Using problem frames with distributed architectures: A case for cardinality on interfaces. In: The Second International Software Requirements to Architectures Workshop (STRAW 2003) (May 2003)
12. Hall, J.G., Rapanotti, L., Jackson, M.: Problem frame semantics for software development. Software and System Modeling 4(2), 189–198 (2005)
13. Hatebur, D., Heise, M.: A UML profile for requirements analysis of dependable software. In: Schoitsch, E. (ed.) Proc. SAFECOMP 2010. LNCS, vol. 6351. Springer, Heidelberg (2010)
14. Hatebur, D., Heisel, M., Schmidt, H.: A pattern system for security requirements engineering. In: Werner, B. (ed.) Proceedings of the International Conference on Availability, Reliability and Security (AReS) IEEE Transactions, pp. 356–365. IEEE, Los Alamitos (2007)
15. Hatebur, D., Heisel, M., Schmidt, H.: A formal metamodel for problem frames. In: Czarnecki, K., et al. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 68–82. Springer, Heidelberg (2008)
16. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley, Reading (2001)
17. Lencastre, M., Botelho, J., Clericuzzi, P., Araújo, J.: A meta-model for the problem frames approach. In: WiSME 2005: 4th Workshop in Software Modeling Engineering (2005)
18. Meszaros, G.: XUnit Test Patterns, Refactoring Test Code. Addison-Wesley, Reading (2007)
19. Seater, R., Jackson, D., Gheyi, R.: Requirement progression in problem frames: deriving specifications from requirements. Requirements Engineering 12(2), 77–102 (2007)
20. Gérard, S., et al.: Papyrus UML Modelling Tool (January 2010), http://www.papyusuml.org/
21. Shaw, M., Garlan, D.: Software Architecture. Perspectives on an Emerging Discipline. Prentice-Hall, Englewood Cliffs (1996)
22. SysML Partners. Systems Modeling Language (SysML) Specification (2005), http://www.sysml.org
23. UML Revision Task Force. OMG Object Constraint Language: Reference (May 2006), http://www.omg.org/docs/formal/06-05-01.pdf
24. UML Revision Task Force. OMG Systems Modeling Language (OMG SysML) (November 2008), http://www.omg.org/spec/SysML/1.1/
25. UML Revision Task Force. OMG Unified Modeling Language: Superstructure (February 2009), http://www.omg.org/docs/formal/09-02-02.pdf
26. UML Revision Task Force. UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE) (November 2009), http://www.omg.org/docs/formal/formal/2009-11-02.pdf
27. van Lamsweerde, A.: From worlds to machines. In: A Tribute to Michael Jackson, Lulu Press (2009)

Practical Parameterised Session Types

Andi Bejleri

Department of Computing, Imperial College London

Abstract. *Parameterised session types* is a type theory studied in the context of *multiparty session types*, that addresses statically the problem of type-safe, deadlock-free interactions in programs of an arbitrary number of processes. The previous work supporting parameterised session types has several shortfalls that limit their utility in practice. We eliminate the shortfalls by introducing a programming idiom of *roles* and a new type system. Roles have the same design as classes in languages such as Java and C#, while the previous model presents an amorphous syntax without concepts on how to incorporate parameterised session types into a mainstream language. The previous model requires programmers to write processes types, in addition to global types, for type-checking, while this model preserves multiparty's lightweight type annotations and type-checking strategy of simply global types. The previous model requires values of parameters to range over finite sets of natural numbers, while this model allows infinite sets of them.

1 Introduction

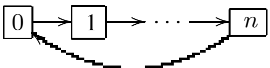
It is well-known that message-based communication constitutes a prime element in the development of applications, namely web services, business protocols, parallel algorithms, multi-core programming, data centers management systems. This has motivated a vast amount of research into techniques, typically type-systems, for guaranteeing communication-safety. Among them, *multiparty session types* [16] is a type theory that addresses statically the problem of type-safe, deadlock-free interactions among a fixed number of processes. *Intuitive syntax of types* and *efficient type-checking strategy* are the main benefits that make multiparty session types stand out from the other systems. A notion of *global type* is introduced through an intuitive syntax to describe the interaction structure between the processes, that is defined by the "sending-receiving" actions in the presence of conditionals and recursion, from a global point of view. Processes are then efficiently validated by type-checking, through the *projection* of global types onto each participant. Whereby, this theory excels when given the number of participants. Unfortunately, in parallel algorithms and other communication-based applications the number of participants is known only at run-time; e.g. in parallel algorithms, the number of processes assigned to compute the answer of a problem instance is in proportion to its size.

Recently, the idea of *parameterised session types* [21] is studied in the context of multiparty session types. With parameterised session types, a global type can describe the communication pattern of an arbitrary number of participants. Communication patterns describe simple and elegant structured interactions in communication based applications. They are used in many parallel computing architectures of parallel algorithms [14], exchange protocols [4] and web-services [20]. Communication patterns, as design

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 270–286, 2010.
© Springer-Verlag Berlin Heidelberg 2010

patterns, help programmers to design more modular and more understandable system architectures. Common communication patterns are Ring, Tree, Mesh and Hypercube.

A global type includes the $\mathbf{R}$ operator from Gödel's theory T [1] to iterate over parameterised causalities that abstract the repetitive behavior of a pattern and to compose sequentially global types, where parameterised causalities are defined over parameterised principals. For example, in the Ring pattern,



the causality that abstracts the communications from 0 to n is $\mathtt{i} \rightarrow \mathtt{i{+}1}{:}\langle U \rangle$ with some abuse of notation, where $0{\leq}i{\leq}n{-}1$ and $n{\geq}1$. Given the number of participants, the $\mathbf{R}$ operator will iterate over the parameterised causality, and then the global type created will be composed with the causality $\mathtt{n} \rightarrow \mathtt{0} : \langle U \rangle$ to complete an instance of the Ring.

The syntax of processes includes also the $\mathbf{R}$ operator to parameterise participants, to iterate over processes that share the same behavior and to compose in parallel processes. For example, in the Ring, there are three kinds of participants: the first one is 0 which sends to the participant on his left (1) and then receives from the last participant (n), the second one is i for $1{\leq}i{\leq}n{-}1$ that receives from the participants on his right ($\mathtt{i{-}1}$) and then sends to the one on his left ($\mathtt{i{+}1}$) and finally, the last participant n, which receives from the participant on his right ($\mathtt{n{-}1}$) and then sends to the first participant (0). The $\mathbf{R}$ operator will iterate over i, returning on each iteration processes that share the same behavior, and then will compose them in parallel with the processes of 0 and n.

Despite, this strong initial work, parameterised session types have several shortfalls that limit their utility in practice. First, the syntax of processes is amorphous, without concepts to design a library that supports communication of mainstream languages. Second, the type system requires *programmers to write the processes types*, in addition to the global type, for type-checking. The coherence of the process types with respect to the global type is ensured by an *equivalence relation* for every value of each parameter present in the global type. The former aspect of the type system increases the programming effort and the latter diverges from the efficient typing strategy of the fundamental work of global session types [16], where the global type is projected onto participants to obtain the types for typechecking of processes. Third, the type system restricts the computing power of programs, allowing values of parameters to range over finite sets of natural numbers, e.g. parameter $n{:}\{m{:}\mathsf{nat} \mid 0 \leq m \leq 1000\}$ is typed by a bounded set of natural numbers. Finite sets are necessary to provide decidability for the type system, as termination of the equivalence algorithm depends on those sets' cardinality. The upper-bound of the set reflects the maximum capacity of the hardware resources to program date, not matching the purpose of innovative dynamic computing platforms such as clouds where hardware resources flex in response to demand.

This paper eliminates these shortfalls by introducing a programming idiom of *roles* and a new type system. Contributions of this work include:

- *A role defines an abstraction of communication's end-points in mobile processes. It is a blueprint that describes the nature of a communication pattern and the behavior that all run-time processes will share.* The syntactic extension is small but yet

provides a similar concept to classes in class-based languages and so, offers a design on how to incorporate parameterised session types into a mainstream language such as Java and C#.

- A static type system that follows the efficient typing strategy and programming methodology of multiparty session types: programmers first define the global type of the intended pattern and then define each role of it; roles are validated through projection of the global type onto the principals by type-checking. We achieve strict global type annotations in programs and efficient type-checking by extending the multiparty's projection algorithm to parameterised principals.
- Values of parameters range over infinite sets of natural numbers. We use infinite sets to provide full computation power of programs that implement parameterised communication patterns.
- Examples that show how this system can represent various communication patterns and control one of the main sources of programming errors in MPI (a message-passing API to program parallel computers). We illustrate the practical utility of our system through real-world examples from parallel algorithms and key distribution protocol.

Section 2 discusses our syntax of roles, illustrated by the Ring example, and operational semantics[1]. Section 3 gives the syntax of global types for parameterised communication patterns, illustrated by the Ring and Tree patterns. Section 4 defines our typing system, and proofs of decidability and subject reduction. Section 5 describes two real-world examples from parallel algorithms and data exchange protocols. Section 6 surveys related work and section 7 concludes with a discussion of possible future work for this system.

2 Roles

Our system extends that of Bettini *et al.* [7], preserving multiparty's programming methodology and typing strategy. Channels are omitted from the syntax of processes [7], serving a simpler type system than the one introduced by Honda *et al.* [16].

Syntax. Figure 1 provides the syntax of our calculus. A program $\lambda n.\mathrm{E}$ is a function from naturals (the number of participants) to roles composed in parallel. Roles in our calculus are second-class constructs; they can not be computed by functions. This contrasts the design of the previous work [21], where both functions and processes are considered as runtime entities and of the same programming-idiom class. Each role defines a scope that includes the subsequent behaviors. The prefix $\bar{u}[\mathtt{p}_0, \mathtt{p}_1, p](y).R$ represents the behavior of the first principal in the list (possibly parameterised) of principals $\mathtt{p}_0, \mathtt{p}_1, p$ and the process of that role initiates a session with the acceptor processes of shape $u[\mathtt{p}](y).R$ of principals $\mathtt{p}_1$ and p. The prefixes represent the abstract notion of session establishment of an arbitrary number of principals in minimal syntax. The sending construct $c!\langle \mathtt{p}, e\rangle; R$ denotes the action of sending a value to participant $\mathtt{p}$; the receiving construct $c?\langle \mathtt{p}, x\rangle; R$ denotes the action of receiving a value from $\mathtt{p}$. A similar notation

[1] For space reasons, we present only part of the formal model definition, related with essential features of the system and contributions of this work. The full definition of the formal model can be found in a companion technical report [5].

E :: = $\lambda n.$E | E $\mathtt{t}$ | R General expressions

R, S ::=	Roles	$\mid (\nu w)R$	Hiding
$\mid \bar{u}[\mathtt{p}_0, \mathtt{p}_1, p](y).R$	Multicast request	$\mid \mathbf{0}$	Inaction
$\mid u[\mathtt{p}](y).R$	Accept	$\mid R \mid S$	Parallel
$\mid c!\langle \mathtt{p}, e\rangle; R$	Value sending	$\mid \mathbf{R}\ S\ \lambda i.\lambda X.R$	Primitive recursion
$\mid c?\langle \mathtt{p}, x\rangle; R$	Value reception	$\mid X$	Process variable
$\mid c \oplus \langle \mathtt{p}, l\rangle; R$	Selection	$\mid R\ \mathtt{t}$	Application
$\mid c\&\langle \mathtt{p}, \{l_i : R_i\}_{i\in I}\rangle$	Branching	$\mid s{:}h$	Queues

$u ::= x \mid a$	Identifiers	$c ::= y \mid s[\hat{\mathtt{p}}]$	Channels
$p ::= \mathtt{p}_1..\mathtt{p}_\mathtt{n} \mid \mathbf{R}\ p\ \lambda i.\lambda X.p'\ \mathtt{t} \mid X$	List of prin.	$h ::= \epsilon \mid m \cdot h$	Queues
$e ::= \mathtt{t} \mid v \mid e\ \mathsf{op}\ e'$	Expressions	$m ::= (\hat{\mathtt{q}},\hat{\mathtt{p}},v) \mid (\hat{\mathtt{q}},\hat{\mathtt{p}},l)$	Msg. in transit
$v ::= a \mid s[\hat{\mathtt{p}}] \mid \mathrm{n} \mid \mathsf{true} \mid \mathsf{false}$	Values	$\hat{\mathtt{p}}, \hat{\mathtt{q}} ::= \hat{\mathtt{p}}[\mathrm{n}] \mid \mathcal{N}$	Value principal

Fig. 1. Syntax for roles and run-time processes

is used in selection-branching of a label, $c \oplus \langle \mathtt{p}, l\rangle; R$ and $c\&\langle \mathtt{p}, \{l_i : R_i\}_{i\in I}\rangle$, where the former selects one of the labels enumerated in I and sends it to the later. $\nu w.R$ restricts w to R. Parallel composition is standard.

We use the $\mathbf{R}$ operator from System T as in the previous work, to parameterise principals, to iterate and compose in parallel roles. The operator composes a lambda expression, $\lambda i.\lambda X.R$, with another expression, S. The recursive operator can be used also inside the definition of a role to iterate over a particular end-point behavior. Iteration takes place when a natural number is applied to a primitive recursion term, $R\ \mathtt{t}$.

The message queue $s : h$ represents ordered messages in transit to model TCP-like asynchronous communications. Identifiers u can be variables or shared names. A list of principals can be constant or parameterised using the $\mathbf{R}$ operator. The mathematical definition of principals list is missing from the previous work, weakening the properties of the type system as we shall see later. Expressions include parametric mathematical expressions (see Section 3), values and operations such as $e = e'$, e and e' and not e. Values are defined over shared names, session channels, naturals and boolean values. Channels denote channel variables or session channels. Session channel $s[\hat{\mathtt{p}}]$ denotes the channel of the participant $\hat{\mathtt{p}}$ in the session s. Messages in queues are defined as triples: sender, receiver and data (value or label). Messages are run-time entities, therefore they are defined over value principals. Value principals are the same as in the previous work, including participants (`Bob`, `Alice`, ...) or indexed principals over naturals (`W`[3], `W`[2][4], ...).

Operational Semantics. Figure 2 gives the operational semantics via the reduction relation $\longrightarrow$. The interesting features of the rules are how they invoke a program, start a session, instantiate roles, iterate over end-point behavior and exchange messages.

The rule [App] invokes a program by replacing the parameter n with the argument n, as it instantiates roles which are parameterised only by n. Rule [Zero] returns the behavior S and defines the last iteration of the $\mathbf{R}$ operator. Rule [Succ] replaces each occurrence of the index i in R with a predecessor of $\mathrm{n}{+}1$ and replaces X with instances of R returned by the other iterations. When R denotes roles, [Succ] instantiates them

$$(\lambda n.E)\ \mathrm{n} \longrightarrow E\{\mathrm{n}/n\} \quad \text{[App]}$$

$$\mathbf{R}\ S\ \lambda i.\lambda X.R\ 0 \longrightarrow S \quad \text{[Zero]}$$

$$\mathbf{R}\ S\ \lambda i.\lambda X.R\ (\mathrm{n}+1) \longrightarrow R\{\mathrm{n}/i\}\{\mathbf{R}\ S\ \lambda i.\lambda X.R\ \mathrm{n}/X\} \quad \text{[Succ]}$$

$$\bar{a}[\hat{\mathrm{p}}_0..\hat{\mathrm{p}}_\mathrm{n}](y_0).R_0 \mid a[\hat{\mathrm{p}}_1](y_1).R_1 \mid ...a[\hat{\mathrm{p}}_\mathrm{n}](y_\mathrm{n}).R_\mathrm{n} \longrightarrow (\nu s)(R_0\{s[\hat{\mathrm{p}}_0]/y_0\} \mid ... \mid R_\mathrm{n}\{s[\hat{\mathrm{p}}_\mathrm{n}]/y_\mathrm{n}\} \mid s:\emptyset) \quad \text{[Link]}$$

$$s[\hat{\mathrm{p}}]!\langle\hat{\mathrm{q}}, v\rangle; R \mid s:h \longrightarrow R \mid s:h\cdot(\hat{\mathrm{p}},\hat{\mathrm{q}},v) \quad \text{[Send]}$$

$$s[\hat{\mathrm{p}}] \oplus \langle\hat{\mathrm{q}}, l\rangle; R \mid s:h \longrightarrow R \mid s:h\cdot(\hat{\mathrm{p}},\hat{\mathrm{q}},l) \quad \text{[Label]}$$

$$s[\hat{\mathrm{p}}]?(\hat{\mathrm{q}}, x); R \mid s:(\hat{\mathrm{q}},\hat{\mathrm{p}},v)\cdot h \longrightarrow R\{v/x\} \mid s:h \quad \text{[Recv]}$$

$$s[\hat{\mathrm{p}}]\&(\hat{\mathrm{q}}, \{l_i : R_i\}_{i\in I}) \mid s:(\hat{\mathrm{q}},\hat{\mathrm{p}},l_{i_0})\cdot h \longrightarrow R_{i_0} \mid s:h \quad (i_0 \in I) \quad \text{[Branch]}$$

Fig. 2. Reduction rules

in each iteration and composes them in parallel. Otherwise R denotes an end-point behavior that [Succ] iterates when the session has been established.

A session is established among processes via shared channels (a) that denote public points of communication. At this point, every role has been instantiated into processes and the computation follows over value principals. The rule [Link] invokes a session between n peers by generating a fresh session channel s to perform a series of communications and the associated empty session queue, and substitutes the channel into the processes scope. The identity of each principal within a session is represented by the label $\hat{\mathrm{p}}$ in the session channel $s[\hat{\mathrm{p}}]$. The n-party synchronisation captures a handshake among the n participants to establish a n-party link in real-world protocols. The [Send] and [Label] rules insert a message in the tail of the session queue. The receiving rule [Recv] removes a value message of the same sender, as the one specified in the receiving construct, from the head of the queue, and substitutes it in the process. The [Branch] rule removes from the queue a label message of the same sender, as the ones specified in the branching construct. The result of the rule is the process following the label.

Ring. The Ring pattern, described in the introduction, consists of $n{+}1$ workers (named by $\mathtt{W}$) where each has exactly two neighbours: the worker $\mathtt{W}[j]$ communicates with the workers $\mathtt{W}[j{-}1]$ and $\mathtt{W}[j{+}1]$ $(1{\leq}j{\leq}\mathrm{n}{-}1)$, with the exception of $\mathtt{W}[0]$ who communicates with $\mathtt{W}[n]$ and $\mathtt{W}[1]$, and $\mathtt{W}[\mathrm{n}]$ with $\mathtt{W}[n{-}1]$ and $\mathtt{W}[0]$. Due to the enumeration of workers in a non-modular arithmetic, the Ring has three distinct roles: *Starter*, represented by $\mathtt{W}[0]$, *Middle*, represented by $\mathtt{W}[j]$, and *Last*, represented by $\mathtt{W}[\mathrm{n}]$. To ensure the presence of all three roles in a session, we set the number of participants to $n{\geq}2$ but we do not set any upper-bound, which would be the case in the previous work [21]. Below, we provide the main program and roles of the Ring:

$$\texttt{def}\ \ W = \mathbf{R}\ \mathtt{W}[n]\ \lambda i.\lambda X.\mathtt{W}[i+2], X\ \ (n-2)$$

$$\mathit{Starter} \triangleq \bar{a}[\mathtt{W}[0], \mathtt{W}[1], W](y).y!\langle\mathtt{W}[1], v\rangle; y?(\mathtt{W}[n], z); R$$

$$\mathit{Middle(i)} \triangleq a[\mathtt{W}[i+1]](y).y?(\mathtt{W}[i], z); y!\langle\mathtt{W}[i+2], z\rangle; R'$$

$$\mathit{Last} \triangleq a[\mathtt{W}[n]](y).y?(\mathtt{W}[n-1], z); y!\langle\mathtt{W}[0], z\rangle; S$$

$$\mathrm{Ring} \triangleq \lambda n.((\mathbf{R}\ \mathit{Starter} \mid \mathit{Last}\ \lambda i.\lambda X.\mathit{Middle(i)} \mid X)\ \ (n-1))$$

where *W* denotes the parameterised list of principals $\mathtt{W}[2], ..., \mathtt{W}[n]$ mathematically represented through the $\mathbf{R}$ operator, and *Starter* and *Last* are parameterised by n and *Middle* by i. *Middle* is composed in parallel with the process variable X that is used as a placeholder of processes generated in each iteration; in the last iteration for $n{=}0$, X will be replaced with processes of *Starter* and *Last*.

Below, we give the implementation of the *Starter* role in Java. The role is naturally implemented in a particular class and so would the other roles, avoiding the MPI style of programming Single Program Multiple Data.

```
public class Starter{
   public Starter(int port_l, String host_r, int port_r){
      // Set up the sockets for the pattern.
      ServerSocket serverSocket = null;
      Socket clientSocket = null;
      PrintWriter out = null;
      BufferedReader in = null;
      try{
          serverSocket = new ServerSocket(port_l);
          clientSocket = new Socket(host_r, port_r);
          out = ...; // Init. the output stream on clientSocket.
          in = ...; // Init. the input stream on serverSocket.
          // Exchange messages with neighbors.
          out.println("1");
          String m = in.readline();
          ... // close streams and sockets, capture exceptions.
   }
}
```

The class `Starter`, similarly to the role, provides a communication abstraction of the session object that will be generated at run-time given the values for the parameters `port_l`, `host_r`, `port_r`. The parameters abstract the neighbors' address and ports, similarly as n in the role definition. The feature that does not match the Java model with that of our calculus is the communication abstraction. Java uses client-server sockets as communication abstraction, while in our calculus, we use session channels s to define communication between a group of processes, including the n-party handshake. Thus, a Java library that supports communication over a group of processes, would provide the proper features to model roles and session channels, therefore the proper framework to implement parameterised session types.

3 Global Types

Global types [16] describe the interaction structure of a fixed number of processes from a global point of view. With parameterised session types, a global type can describe the communication pattern of an arbitrary number of participants.

3.1 Global Types for Parameterised Communication Patterns

Figure 3 gives the syntax of global types. A message of type U is exchanged between two principals, $\mathtt{p} \to \mathtt{p}' : \langle U \rangle.G$, where $\mathtt{p}$ and $\mathtt{p}'$ are respectively the sender and the

$G ::=$	Global types	$\mid \mathbf{R}\, G\, \lambda i.\lambda \mathbf{x}.G'$	Primitive recursion
$\mid \mathtt{p} \to \mathtt{p}' : \langle U \rangle.G$	Message	$\mid \mathbf{x}$	Primitive rec. type var.
$\mid \mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i \in I}$	Branching	$\mid G\ \mathtt{t}$	Application
$\mid \mu \mathbf{t}.G$	Recursion	$\mid \mathsf{end}$	End
$\mid \mathbf{t}$	Rec. type var.		

$\mathtt{p} ::= \mathtt{p}[\mathtt{i}] \mid \mathcal{N}$	Principals	$\mathcal{N} ::= \mathtt{Alice} \mid \mathtt{Worker} \mid \ldots$	Participants
$\mathtt{i} ::= \mathtt{t} \mid i \mid \mathrm{n} * \mathtt{i} \mid \mathtt{t} \pm \mathtt{i}$	Index expr.	$U ::= V \mid T$	Message type
$\mathtt{t} ::= \mathrm{n} \mid n \mid \mathtt{t}\ \mathrm{op}\ \mathrm{n} \mid \mathrm{n}^{\mathtt{t}}$	Par. expr.	$V ::= \mathsf{bool} \mid \mathsf{nat} \mid .. \mid \langle G \rangle$	Value type

Fig. 3. Global types

receiver. Branching is defined over labels which identify the paths of a conversation, $\mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i \in I}$; i.e. participant p internally chooses one of the labels l_i enumerated by I and then sends it to participant $\mathtt{p}'$ and the conversation follows G_i. Infinite behavior is represented by recursively defined global types $\mu \mathbf{t}.G$. end signifies the end of a conversation.

The $\mathbf{R}$ operator is added to the syntax of global types to describe communication patterns of an arbitrary number of principals, as in the previous work. The parameters that abstract the number of participants are bound by the binders in the lambda expressions of roles, since both global type and roles are part of the program definition. This contrasts the design of the previous work where a special binder Π is introduced in the global type syntax, prohibiting syntactically the relation between the number of participants in global types and programs. Throughout the paper we will refer to primitive recursive global types as *product global types*, as they abstract all instances of the parameterised global type. The infinite set of instances generated from the $\mathbf{R}$ operator can be understood through the two reduction rules:

$$\mathbf{R}\, G\, \lambda i.\lambda \mathbf{x}.G'\ 0 \longrightarrow G$$
$$\mathbf{R}\, G\, \lambda i.\lambda \mathbf{x}.G'\ (\mathrm{n}+1) \longrightarrow G'\{\mathrm{n}/i\}\{(\mathbf{R}\, G\, \lambda i.\lambda \mathbf{x}.G'\ \mathrm{n})/\mathbf{x}\}$$

For each natural, we obtain a global type by applying the two rules. In each iteration, the index variable in G' is substituted by a predecessor of $n{+}1$ and $\mathbf{x}$ is replaced by instances of the parameterised causalities present in G', except 0 when $\mathbf{x}$ is replaced by instances of G.

Principals $\mathtt{p}, \mathtt{p}', \mathtt{q}$, .. include primitive participants Alice, Bob, ... and indexed principals defined over one or multiple index expressions $\mathtt{W}[i]$, $\mathtt{W}[i{+}1][j{+}1]$, ... Index expressions $\mathtt{i}$ are represented by parametric linear functions, where n ranges over naturals, i ranges over index variables and $\mathtt{t}$ ranges over parametric expressions. Parametric expressions range over variables n, naturals n, arithmetical operations ($\mathtt{t}{+}\mathrm{n}$, $\mathtt{t}{-}\mathrm{n}$, $\mathtt{t}{*}\mathrm{n}$) and exponentiation of base natural. The index expressions of the previous work [21] are more general than in this system, including a more sophisticated mathematical definition and more than one index variable per index expression. This expressivity comes at the cost of having values of parameters to range over finite sets of naturals in the conservative type system. Our design of index expressions as parametric linear functions comes from the observation that the information flow follows a straight line, neither a curve nor other forms of line, in the patterns/virtual-topologies we have studied so far, namely Ring, Star, Tree and Mesh. For simplicity and without reducing the practical

$T ::= !\langle \mathtt{p}, U\rangle; T$	Output	$\mid$	$\mathbf{R}\ T\ \lambda i.\lambda \mathbf{x}.T'$	Primitive Recursion
$\mid\ ?\langle \mathtt{p}, U\rangle; T$	Input	$\mid$	$T\ \mathtt{t}$	Application
$\mid\ \oplus\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle$	Selection	$\mid$	$\mathbf{x}$	Primitive Recursion Variable
$\mid\ \&\langle \mathtt{p}, \{l_i : T_i\}_{i\in I}\rangle$	Branching	$\mid$	$\mathbf{t}$	Recursion Variable
$\mid\ \mu\mathbf{t}.T$	Recursion	$\mid$	end	End

Fig. 4. Role types

expressiveness of our system, we have designed index expression to have at most one parameter n. A type U ranges over primitive (bool, nat, ...) and global types ($\langle G\rangle$), and role types (T) (see Figure 4).

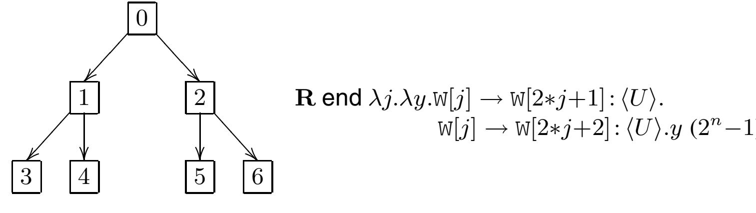
3.2 Ring and Tree Communication Patterns

We illustrate how the formal model of this work can represent various communication patterns such as Ring and Tree.

Ring pattern The global type of the Ring, described in the introduction and Section 2, is defined below. The causality $\mathtt{W}[n{-}j{-}1] \to \mathtt{W}[n-j] : \langle U\rangle$ abstracts the repetitive behavior of the pattern from 0 to n, while $\mathtt{W}[n] \to \mathtt{W}[0] : \langle U\rangle$ completes the Ring pattern. The type specifies that the first message is sent by $\mathtt{W}[0]$ to $\mathtt{W}[1]$ for $j{=}n{-}1$, and the last one is sent by $\mathtt{W}[\mathrm{n}]$ back to $\mathtt{W}[0]$ for $n{=}0$.

$$\begin{array}{l}\mathbf{R}\ \mathtt{W}[n] \to \mathtt{W}[0] : \langle U\rangle.\mathsf{end}\\ \quad \lambda j.\lambda \mathbf{y}.\mathtt{W}[n{-}j{-}1] \to \mathtt{W}[n-j] : \langle U\rangle.\mathbf{y}\ n\end{array}$$

Tree pattern. The Tree pattern consists of $2^{n+1}{-}1$ workers organized in a binary tree. The global type below specifies a message exchange between a parent and its children.



$$\begin{array}{l}\mathbf{R}\ \mathsf{end}\ \lambda j.\lambda y.\mathtt{W}[j] \to \mathtt{W}[2{*}j{+}1] : \langle U\rangle.\\ \qquad\qquad \mathtt{W}[j] \to \mathtt{W}[2{*}j{+}2] : \langle U\rangle.y\ (2^n{-}1)\end{array}$$

A tree has three kinds of nodes: root, internal and leaf. The principal running on the root sends a message to its children; the ones on internal nodes send a message to their children and receive a message from their parents; the ones on leaf nodes receive a message from their parents. The three kind of nodes define three distinct roles of the Tree. An internal or leaf node is enumerated by an even or odd number, and thus the mathematical expressions that identify the parent and children of each of these nodes are different. For this reason, even and odd nodes define two distinct roles in the same kind of node, internal/leaf. Thus, we have distinguished five roles in the Tree: *Root* represented by $\mathtt{W}[0]$, *OddIn* and *EvenIn* by $\mathtt{W}[2{*}i{+}1]$ and $\mathtt{W}[2{*}i{+}2]$ $(0{\le}i{\le}2^{n-1}{-}2)$, and, *OddLeaf* and *EvenLeaf* by $\mathtt{W}[2{*}i{+}1]$ and $\mathtt{W}[2{*}i{+}2]$ $(2^{n-1}{-}1{\le}i{\le}2^n{-}2)$. To ensure the presence of all five roles in a session, we set $n \ge 2$. We only provide three of the Tree's roles, relegating the other ones and the main program in [5]:

$Root \triangleq \bar{a}[\mathtt{W}[0], \mathtt{W}[1], \mathtt{W}](y).y!\langle \mathtt{W}[1], f(1)\rangle; y!\langle \mathtt{W}[2], f(2)\rangle; R'$
$OddIn(i) \triangleq a[\mathtt{W}[2{*}i{+}1]](y).y!\langle \mathtt{W}[4{*}i{+}3], f(4{*}i{+}3)\rangle; y!\langle \mathtt{W}[4{*}i{+}4], f(4{*}i{+}4)\rangle; y?(\mathtt{W}[i], z); R$
$OddLeaf(i) \triangleq a[\mathtt{W}[2^n{-}1{+}2{*}i]](y).y?(\mathtt{W}[2^{n-1}{-}1{+}i], z); S$

where f is a function from naturals to U. It is interesting to note that index calculation in the principals of the global type is less complex than in the ones of the roles. This is a direct advantage of the global representation of interactions. The type system of this work statically ensures that the principals in the role's actions are the same to the ones specified in the causalities of global types; e.g. for role *OddIn*, the type system ensures that the first message is sent to $\mathtt{W}[4{*}i{+}3]$. The problem of index calculation in the roles of parallel algorithms has been recognized also by the MPI community [14] as a source of program errors.

4 Type System

This section describes our extension of multiparty's static type system to support parameterised sessions. An essential aspect is the preservation of multiparty's lightweight type annotations and efficient typing strategy of simply global types.

4.1 Projection, Ordering and R-Elimination

Projection. A global type's projection onto the principals of roles produces types (See Figure 4) that capture the behavior of roles. The given global type is defined on well-formed indexed principals and parametric expressions are applied only to product global types, ensured by the kinding judgment $\Theta; C \vdash G \blacktriangleright \kappa$ (see [5]). A well-formed principal, ensured by the judgment $C \vdash \mathtt{p}$, is either a participant or an indexed principal where the set of values of each index expression is defined over naturals.

Definition 4.1. *Given global type G, principal $\mathtt{q}$, and the context C of parameter variables present in G and $\mathtt{q}$, and index variables present in $\mathtt{q}$, if $\emptyset; C \vdash G \blacktriangleright \kappa$ and $C \vdash \mathtt{q}$ then the projection of G onto $\mathtt{q}$, denoted $G \upharpoonright \mathtt{q}$, is defined inductively on G:*

$$\mathtt{p} \to \mathtt{p}' : \langle U\rangle.G \upharpoonright \mathtt{q} = \begin{cases} !\langle \mathtt{p}'\{\mathtt{p}=\mathtt{q}\}, U\rangle(\mathtt{p}); ?\langle \mathtt{p}\{\mathtt{p}'=\mathtt{q}\}, U\rangle(\mathtt{p}'); (G \upharpoonright \mathtt{q}) & \text{if } C \vdash \mathtt{p}=\mathtt{q} \text{ and } C \vdash \mathtt{p}'=\mathtt{q},\\ !\langle \mathtt{p}'\{\mathtt{p}=\mathtt{q}\}, U\rangle(\mathtt{p}); (G \upharpoonright \mathtt{q}) & \text{if } C \vdash \mathtt{p}=\mathtt{q},\\ ?\langle \mathtt{p}\{\mathtt{p}'=\mathtt{q}\}, U\rangle(\mathtt{p}'); (G \upharpoonright \mathtt{q}) & \text{if } C \vdash \mathtt{p}'=\mathtt{q},\\ G \upharpoonright \mathtt{q} & \text{otherwise}\end{cases}$$

$$\mathtt{p} \to \mathtt{p}' : \{l_i : G_i\}_{i\in I} \upharpoonright \mathtt{q} = \begin{cases} \oplus\langle \mathtt{p}'\{\mathtt{p}=\mathtt{q}\}, \{l_i : \&\langle \mathtt{p}\{\mathtt{p}'=\mathtt{q}\}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}\rangle(\mathtt{p}') \\ \qquad\qquad \}_{i\in I}\rangle(\mathtt{p}) & \text{if } C \vdash \mathtt{p}=\mathtt{q} \text{ and } C \vdash \mathtt{p}'=\mathtt{q},\\ \oplus\langle \mathtt{p}'\{\mathtt{p}=\mathtt{q}\}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}\rangle & \text{if } C \vdash \mathtt{p}=\mathtt{q},\\ \&\langle \mathtt{p}\{\mathtt{p}'=\mathtt{q}\}, \{l_i : G_i \upharpoonright \mathtt{q}\}_{i\in I}\rangle & \text{if } C \vdash \mathtt{p}'=\mathtt{q},\\ \sqcup_{i\in I} G_i \upharpoonright \mathtt{q} & \text{if } C \nvdash \mathtt{p}=\mathtt{q}, C \nvdash \mathtt{p}'=\mathtt{q} \\ & \forall i,j \in I. G_i \upharpoonright \mathtt{q} \bowtie G_j \upharpoonright \mathtt{q}\end{cases}$$

$\mu\mathbf{t}.G \upharpoonright \mathtt{q} = \mu\mathbf{t}.(G \upharpoonright \mathtt{q}) \quad \mathbf{t} \upharpoonright \mathtt{q} = \mathbf{t} \quad \mathtt{end} \upharpoonright \mathtt{q} = \mathtt{end}$

$\mathbf{R}\ G\ \lambda i.\lambda \mathbf{x}.G' \upharpoonright \mathtt{q} = \mathbf{R}\ (G \upharpoonright \mathtt{q})\ \lambda i.\lambda \mathbf{x}.(G' \upharpoonright \mathtt{q}) \quad \mathbf{x} \upharpoonright \mathtt{q} = \mathbf{x} \quad G\ \mathtt{t} \upharpoonright \mathtt{q} = (G \upharpoonright \mathtt{q})\ \mathtt{t}$

Projection is intuitive and holds some of the technical challenges of this system, which we discuss in the following paragraphs. In the role types returned, the principal in brackets attached to an action denotes the principal that performs that action, and is used to sort actions and eliminate the $\mathbf{R}$ operator from role types as we shall see later. The equality between a global type principal $\mathtt{p}$ and role principal $\mathtt{q}$ is defined as a relation $\vdash \mathtt{p}{=}\mathtt{q}$ over the context C, which ensures that the set of values of $\mathtt{p}$ is a subset of the set of values of $\mathtt{q}$. For space's sake, we relegate the formal definition of the relation to [5]. The intuition underlying this design originates from the knowledge that an action performed by every process of the same role is captured by the same causality in the global type.

In product global types, an indexed principal can appear in both sides of a parameterised causality for different values of the index variable. This occurrence is covered by the first case of projection for message exchange and branching.

The index variables of principals in global types are different from the ones in roles, as they are bound by different binders. For this reason, we need to translate the role types being expressed from global type indexes to role ones. The $\mathtt{p}'\{\mathtt{p} = \mathtt{q}\}$ operation substitutes the index variables in $\mathtt{p}'$ with expressions in terms of indexes of $\mathtt{q}$, obtained by the relation $\mathtt{p} = \mathtt{q}$ where $\mathtt{p}$ and $\mathtt{p}'$ have the same index variables.

In branching, in the case when $\mathtt{q}$ is not equal neither to $\mathtt{p}$ nor to $\mathtt{p}'$, all inductive projections of $\mathtt{q}$ should return an identical role type up to mergeability $\bowtie$. The notion of mergeability is introduced in [12] as an equivalence relation over role types. Intuitively, two different $\&$ role types are mergeable if denoted by different labels; e.g. the projection of global type $\mathtt{W}[1] \to \mathtt{W}[2] : \begin{cases} \mathit{true} : \mathtt{W}[2] \to \mathtt{W}[3] : \{\mathit{true} : G, \\ \mathit{false} : \mathtt{W}[2] \to \mathtt{W}[3] : \{\mathit{false} : G' \end{cases}$ onto $\mathtt{W}[3]$ returns $\&\langle \mathtt{W}[2], \{\mathit{true}{:}G \restriction \mathtt{W}[3], \mathit{false}{:}G' \restriction \mathtt{W}[3]\}\rangle$, where $G \restriction \mathtt{W}[3] \neq G' \restriction \mathtt{W}[3]$.

Proposition 4.2. *The relation $C \vdash \mathtt{p} = \mathtt{q}$ is decidable.*

Theorem 4.3. *The projection of a global type onto principals is decidable.*

Proof. Straightforward from Proposition 4.2.

Ordering and R-elimination. Actions in the role types, returned by projection, are sorted to preserve the order of appearance in all instances of a parameterised global type. We can note from the first case of projection in the message global type, that the order of actions is not preserved; i.e., the sending action is always placed before the receiving one. However, the appearance order of actions is not broken only in the projection of a causality, but also in the sequential composition of other actions returned by projection. The reason behind this is that the order of actions depends on the order of principals performing those actions.

Definition 4.4. *The appearance order relation between two actions ($\mathtt{order}$) is defined as the appearance order of the principals performing those actions:*

$$\mathtt{order}(!/?[2]\langle \mathtt{p}_1, U\rangle(\mathtt{p}_1'), !/?\langle \mathtt{p}_2, U'\rangle(\mathtt{p}_2')) \ \textit{iff}\ \mathtt{order}(\mathtt{p}_1', \mathtt{p}_2')$$
$$\mathtt{order}(\oplus/\&\langle \mathtt{p}_1, \{l_i{:}T_i\}_{i\in I}\rangle(\mathtt{p}_1'), \oplus/\&\langle \mathtt{p}_2, \{l_i{:}T_i'\}_{i\in I'}\rangle(\mathtt{p}_2')) \ \textit{iff}\ \mathtt{order}(\mathtt{p}_1', \mathtt{p}_2')$$

[2] !/? denotes either ! or ?.

Definition 4.5. *The appearance order between principals is defined as a lexicographical order over the index expressions that define them:*

$\mathtt{order}(\mathcal{N}[\mathtt{i}_1]...[\mathtt{i}_i]...[\mathtt{i}_n], \mathcal{N}[\mathtt{i}'_1]...[\mathtt{i}'_i]...[\mathtt{i}'_n])$ *iff* $\mathtt{order}(\mathtt{i}_i, \mathtt{i}'_i)$ *for* $1 \leq i \leq n$ *and* $\forall j.1 \leq j \leq i-1.\ C \vdash \mathtt{i}_j = \mathtt{i}'_j$ *and* $C \nvdash \mathtt{i}_i = \mathtt{i}'_i$,

where the appearance order between index expressions in their canonical form is defined as:

$\mathtt{order}(\mathtt{t}{-}\mathrm{n}{*}i, \mathtt{t}'{-}\mathrm{n}'{*}i)$ *iff* $C \vdash \mathtt{t}{-}\mathrm{n}{*}i \geq \mathtt{t}'{-}\mathrm{n}'{*}i$ *and*

$\mathtt{order}(\mathtt{t}{+}\mathrm{n}{*}i, \mathtt{t}'{+}\mathrm{n}'{*}i)$ *iff* $C \vdash \mathtt{t}{+}\mathrm{n}{*}i \leq \mathtt{t}'{+}\mathrm{n}'{*}i$.

The order of index expression is defined on the basis that the value of i decreases in each iteration of the $\mathbf{R}$ global type, resulting in the increase of values for expressions $\mathtt{t}{-}\mathrm{n}{*}i$ and the decrease for $\mathtt{t}{+}\mathrm{n}{*}i$. Thus, in two expressions of the form $\mathtt{t}{-}\mathrm{n}{*}i$, a value will appear first in the bigger expression for bigger value of i and then in the smaller one for smaller value of i. And, in two expressions of the form $\mathtt{t}{+}\mathrm{n}{*}i$, a value will appear first in the smaller expression for bigger value of i and then in the bigger one for smaller value of i. No ordering can be defined for expressions of opposite monotonicity, e.g. $\mathtt{t}{-}\mathrm{n}{*}i$ and $\mathtt{t}'{+}\mathrm{n}'{*}i$, as some values will appear first in the former and second in latter, whilst some others vice versa.

The $\mathbf{R}$ operator in global types iterates over parameterised causalities and defines repetitive behavior for non index-parameterised principals. For these principals, we keep the $\mathbf{R}$ operator and the argument applied in the role types, otherwise we eliminate it by composing the two sub-types, and then later the argument. The R-elimination function is denoted by ξ in the typing rules, formally defined in [5].

4.2 Typing Rules

Figure 5 describes the program typing rules. Γ maps shared names, process names and type variables to types, while τ represents channel and product types, defined as:

$$\tau ::= \Delta \mid \Pi n{:}\mathtt{T}.\tau \mid \Pi i{:}\mathtt{I}.\tau \qquad \Delta ::= \emptyset \mid \Delta, c{:}T \qquad \Gamma ::= \emptyset \mid \Gamma, u{:}S \mid \Gamma, \mathbb{X}{:}S\ T \mid \Gamma, X{:}\Delta$$

The rules of appealing interest are those for program and session initiation. Rule $\lfloor$TFUN$\rfloor$ augments the context C with mapping for parameter variables and ensures that the subterm is typed. Rule $\lfloor$TAPPF$\rfloor$ checks if the argument applied to the lambda abstraction falls in the set of values $\mathtt{T}$, where $\min(\mathtt{T})$ represents the minimum value n. For primitive recursion, we ensure that the sub-terms are well-typed in the augmented contexts Γ and C. If primitive recursion specifies a repetitive behavior of a role, then $\Delta\ 0$ and $\Delta\ i{+}1$ return the sub-role type for type-checking of the respective sub-terms. Otherwise, $\Delta\ 0$ and $\Delta\ i{+}1$ return Δ. The definition of this rule is similar to that of the previous work, but it does not contain index substitution, simplifying proofs of the properties. The rule of applying a parametric expression to primitive recursion is similar to $\lfloor$TAPPF$\rfloor$, but it also ensures that the argument applied is a successor of the biggest index value. Roles are type-checked by the role types, returned by projection, sorting and $\mathbf{R}$-elimination. The previous work's typechecking algorithm [21] uses the processes types written by programmmers to type-check the processes. The coherence of processes types with respect to global types is proved by an equivalence algorithm that uses the sets of parameters' values to produce instances of product global types

$$\frac{\Gamma; C, n : \mathtt{T} \vdash E \triangleright \tau}{\Gamma; C \vdash \lambda n.E \triangleright \Pi n{:}\mathtt{T}.\tau}\ \lfloor \textsc{TFun} \rfloor \qquad \frac{\Gamma; C \vdash E \triangleright \Pi n{:}\mathtt{T}.\tau \quad C \vdash \mathtt{t} \geq \mathsf{min}(\mathtt{T})}{\Gamma; C \vdash E\ \mathtt{t} \triangleright \tau}\ \lfloor \textsc{TAppF} \rfloor$$

$$\frac{\begin{array}{c}\Gamma; C \vdash S \triangleright \Delta\ 0 \\ \Gamma, X : \Delta\ i; C, i : \mathtt{I} \vdash R \triangleright \Delta\ i+1\end{array}}{\Gamma; C \vdash \mathbf{R}\ S\ \lambda i.\lambda X.R \triangleright \Pi i{:}\mathtt{I}.\Delta}\ \lfloor \textsc{TPRec} \rfloor \qquad \frac{\begin{array}{c}C \vdash \mathtt{t} \quad \Gamma; C \vdash R \triangleright \\ \Pi i{:}\{i\ |0 \leq i \leq \mathtt{t}-1\}.\Delta\end{array}}{\Gamma; C \vdash R\ \mathtt{t} \triangleright \Delta\ \mathtt{t}}\ \lfloor \textsc{TAppR} \rfloor$$

$$\frac{\begin{array}{c}\Gamma \vdash u : \langle G \rangle \quad \emptyset; C \vdash G \blacktriangleright \mathsf{Type} \\ C \vdash \mathtt{p}_0, \mathtt{p}_1, p \quad C \vdash \mathsf{pid}(G) = \{\mathtt{p}_0, \mathtt{p}_1, p\} \\ \Gamma; C \vdash R \triangleright \Delta, y : \xi(G \upharpoonright \mathtt{p}_0)\end{array}}{\Gamma; C \vdash \bar{u}[\mathtt{p}_0, \mathtt{p}_1, p](y).R \triangleright \Delta}\ \lfloor \textsc{TAcc} \rfloor \qquad \frac{\begin{array}{c}\emptyset; C \vdash G \blacktriangleright \mathsf{Type} \\ \Gamma \vdash u : \langle G \rangle \quad C \vdash \mathtt{p} \\ \Gamma; C \vdash R \triangleright \Delta, y : \xi(G \upharpoonright \mathtt{p})\end{array}}{\Gamma; C \vdash u[\mathtt{p}](y).R \triangleright \Delta}\ \lfloor \textsc{TReq} \rfloor$$

$$\frac{\Gamma; C \vdash e \triangleright S \quad \Gamma \vdash R \triangleright \Delta, c : T}{\Gamma; C \vdash c!\langle \mathtt{p}, e \rangle; R \triangleright \Delta, c :!\langle \mathtt{p}, S \rangle; T}\ \lfloor \textsc{TOut} \rfloor \qquad \frac{\Gamma, x : S; C \vdash R \triangleright \Delta, c : T}{\Gamma; C \vdash c?\langle \mathtt{p}, x \rangle; R \triangleright \Delta, c :?\langle \mathtt{p}, S \rangle; T}\ \lfloor \textsc{TIn} \rfloor$$

Fig. 5. Program and role typing

and processes types. Then, the instances of processes type are checked if they are the same to the one returned from projection of the global type instance, using multiparty's projection algorithm [16]. All the conditions to invoke projection are ensured by $\lfloor \textsc{TAcc} \rfloor$ and $\lfloor \textsc{TReq} \rfloor$. Rule $\lfloor \textsc{TAcc} \rfloor$ checks also that the set of principals, present in the session, is the same to the one of global type. The equality relation between the two sets of principals is defined over the mathematical definition of parameterised list of principals discussed in Section 2, missing from the previous work.

Rules $\lfloor \textsc{TOut} \rfloor$ and $\lfloor \textsc{TIn} \rfloor$ ensure that the sub-terms are typed and check if the principal in the primitives is the same as the one in the role-types. Other standard rules lookup for type variables in Γ, and type branching, hiding, inaction and parallel composition.

Properties. In this paragraph, we state type preservation for the formal system presented in this paper. The full proof is given in a companion technical report [5].

Theorem 4.6 (Type Preservation). *If $\Gamma; C \vdash E \triangleright \tau$, and $E \longrightarrow E'$, then there exists τ' where $\tau \Rightarrow \tau'$, such that $\Gamma; C \vdash E' \triangleright \tau'$.*

Proof. By induction over the derivation of $E \longrightarrow E'$. The proof relies on standard substitution lemmas. Reduction of types $\tau \Rightarrow \tau'$ reflects reduction of processes in presence of application, and sending and receiving of values/labels.

Although, we do not have a formal proof, we believe the system satisfies the standard progress property. Indeed, our system benefits the proof of progress for well-typed processes willing to start a session from Bettini *et al.* [7]. To complete the proof, we need to ascertain that a well-typed program reduces to the above processes and that a well-typed iterative behavior reduces further. We leave the proof of progress for future work. The previous system provides progress for the formal model.

5 Real-World Examples

Jacobi Solution of the Discrete Poisson Equation [14]. Poisson's equation is widely used in many areas of the natural sciences. Jacobi's method converges on a solution by repeatedly replacing each element of the input grid by an adjusted average of its four neighbouring values. The grid can be divided up and the algorithm is performed on each subgrid in separate processes. Neighbouring processes must exchange their subgrid boundary values (ghost-points) as they are updated. We illustrate a two-dimensional (mesh) decomposition of the grid into $n{*}m$ processes, where $n, m{\geq}2$. The process on the (n, m) subgrid, top right corner, controls the termination condition for all processes and sends the first message in the mesh. The global type for the said interactions is:

$$\mathit{Jacobi} \triangleq \mu\mathbf{t}.\mathtt{W}[n][m] \to \mathtt{W}[n][m-1], \mathtt{W}[n-1][m] : \{\mathtt{true} : \mathit{iterate}, \mathtt{false} : \mathit{return}\}.$$

The stopping condition is propagated in the processes following the pattern of the diagram below. Next to it, the global type (*iterate*) for propagating the `true` label.

1. $\mathbf{R}$
2. $\mathbf{R}\ \ (\mathbf{R}\ \mathit{ghst–pnt}\ \lambda k{:}\mathtt{I}'.\lambda\mathbf{z}.\mathtt{W}[0][k{+}1] \to \mathtt{W}[0][k]{:}\{\mathtt{true}{:}\mathbf{z}\}\ m)$
3. $\quad\lambda i : \mathtt{I}.\lambda\mathbf{x}.$
4. $\qquad(\mathbf{R}\ \mathtt{W}[i{+}1][m] \to \mathtt{W}[i][m]{:}\{\mathtt{true}{:}\mathbf{x}\}$
5. $\qquad\quad\lambda j{:}\mathtt{I}'.\lambda\mathbf{y}.\mathtt{W}[i{+}1][j{+}1] \to \mathtt{W}[i{+}1][j]{:}\{\mathtt{true}{:}\mathbf{y}\}$
6. $\qquad\quad m)$
7. $\quad n{-}1$
8. $\lambda l{:}\mathtt{I}.\lambda\mathbf{w}.\mathtt{W}[n][l{+}1] \to \mathtt{W}[n][l]{:}\{\mathtt{true}{:}\mathbf{w}\}\ (m{-}1)$

Propagation of the label in the top row, is described in the causality of line 8, in all the rows, except top and bottom, line 5, in the leftmost column in line 4 and in the bottom row, line 2.

Each process maintains a copy of the boundary values of its neighbours and exchanges them on each iteration of the algorithm. The diagram below portrays how these values are exchanged between the processes, followed by the global type (*ghst–pnt*).

1. $\mathbf{R}\ \ (\mathbf{R}\ \mathit{conv–data}\ \lambda k{:}\mathtt{I}'.\lambda\mathbf{z}.\mathtt{W}[0][k+1] \leftrightarrow \mathtt{W}[0][k]{:}\langle U\rangle\ m)$
2. $\quad\lambda i{:}\mathtt{I}.\lambda\mathbf{x}.$
3. $\qquad(\mathbf{R}\ \mathtt{W}[i+1][0] \leftrightarrow \mathtt{W}[i][0]{:}\langle U\rangle.\mathbf{x}$
4. $\qquad\quad\lambda j{:}\mathtt{I}'.\lambda\mathbf{y}.\mathtt{W}[i+1][j+1] \leftrightarrow \mathtt{W}[i][j+1]{:}\langle U\rangle.$
5. $\qquad\qquad\qquad\mathtt{W}[i+1][j+1] \leftrightarrow \mathtt{W}[i+1][j]{:}\langle U\rangle.\mathbf{y}$
6. $\qquad m)$
7. $\quad n$

$\mathtt{p} \leftrightarrow \mathtt{p}'{:}\langle U\rangle$ is a shortcut for $\mathtt{p} \to \mathtt{p}'{:}\langle U\rangle.\mathtt{p}' \to \mathtt{p}{:}\langle U\rangle$. The exchange of ghost-points in all the rows and columns, except the leftmost column line 3 and bottom row line 1, is described in the causalities of line 4 and 5. The full definition of the global type and roles is given in [5].

Group Diffie-Hellman with Complete Key Authentication Protocol [4]. The Diffie-Hellman protocol is used in password-authentication key agreement and public key infrastructure. Every group M_i $(0{<}i{<}n)$ generates and encrypts a random exponent, that together with the data received from M_{i-1} is then sent to M_{i+1}. Lastly, M_n receives

the data from M_{n-1}, computes the group key and broadcasts it to all other parties. The global type of the protocol is defined as:

$$\mathbf{R}\ (\mathbf{R}\ \mathsf{end}\ \lambda k : \mathtt{I}.\lambda \mathbf{z}.\mathtt{M}[n] \rightarrow \mathtt{M}[k] : \langle key \rangle.\mathbf{z}\ n) \text{ //Broadcasting the group key}$$
$$\lambda i : \mathtt{I}.\lambda \mathbf{x}.\mathtt{M}[n-i-1] \rightarrow \mathtt{M}[n-i] : \langle data \rangle.\mathbf{x}\ n \text{ //Exchanging data on a line pattern}$$

This protocol is modelled in a system of contracts [13]. In that model, an extra private channel is used by the last group M_n to send the key to every other group. The private channel is forwarded between the groups through delegation. A condition is added to the protocol description to check whether the key is sent to every group or not.

In our model, we do not need an extra private channel to send the key, as communications between parties of a session are always defined over private channels. Also, we do not need to add a condition that checks whether the key is sent to every group as this is granted by the semantics of the $\mathbf{R}$ operator.

6 Related Work

The idea of parameterised session types originated from our previous research on investigating the expressivity of session types for parallel algorithms [6]. This idea has been modelled recently in our work [21], which differences with this system were discussed in the introduction and throughout the paper.

Our formal system is modelled after Bettini *et al.* [7], a simpler version of Honda *et al.* [16], which difference was discussed at the beginning of Section 2; none of these systems are expressive enough to model parameterised communication patterns. The $\mathbf{R}$ operator used in the initial and present work was introduced by Gödel in System *T* [1]. The idea of using the $\mathbf{R}$ operator comes from Nelson's work on adding primitive recursion to the lambda calculus [17]. As a result, his system can type functions previously untyped in ML. Our use of the $\mathbf{R}$ operator models parameterised sessions.

Session types were first introduced by Honda *et al.* [15,19] to capture the interaction structure of two processes. Their type system checks whether for each "send" on one process corresponds a "receive" on the other and vice versa. Honda *et al.* [16] extended their system from two-parties to n-parties. Using an intuitive syntax, they introduced a notion of global type to describe the interaction structure of n processes from a global viewpoint. Multiparty session types have been studied also by Bonelli and Compagnoni [8]. Their type system is defined over binary session types, obtained by projecting processes local types onto principals. Session types have been used to type service-oriented multiparty communications [9]. The calculus proposed permits communications inside and outside a session to model merging of two running sessions. Type safety and progress properties are not provided for the formal model.

Contracts [13] are another typing model of mobile processes, defined over processes as behavioral types and not over channels as session types [7]. Consequently, they can well-type more correct programs than session types. However, the expressiveness of the type system comes at a practical cost. Contracts have no intuitive syntax as global types and no iterative construct as in our system. Thus, they do not provide a practical model to design a programming language that supports communication and elegantly expresses parameterised communication patterns; e.g. the key exchange protocol (Sec. 5) has been augmented with additional interactions to check the end of a send-iteration.

The conversation calculus [11] is based on boxed ambients [10] and not in the π-calculus as session types are. Typing is similar to the one of contracts and thus the system carries the same disadvantages when compared to session types. The calculus models dynamic joining and leaving of participants within a session.

Behaviors [2,18] describe the communication behaviour that captures the causal constrains of a concurrent program through terms of a process algebra, similarly to contracts. An implementation [3] of their system for deriving behaviors is provided for CML programs, where their notion of communication pattern, expressed using behaviors, is similar to our notion of role type. Behaviors have a similar typing discipline as contracts, thus they lack the same features of session types as contracts do.

7 Conclusions and Future Work

This paper presented a practical design of parameterised session types. The idiom of roles has the same design as classes in class-based languages, offering a practical concept on how to incorporate parameterised session types into mainstream languages such as Java and C#. This contrasts with the amorphous design of the previous work of parameterised session types. The static type system here follows the efficient typing strategy and programming methodology of multiparty session types: programmers first define the global type of the intended pattern and then define each role of it; roles are then validated through projection of the global type onto the principals by type-checking. This contrasts with the heavyweight type annotated programs in the previous work, where programmers write the processes types, in addition to the global type, for type-checking. Also, the coherence of the processes types with respect to the global type is provided through a type equivalence relation. The system here allows values of parameters to range over infinite sets of naturals to provide full computation power of programs that implement parameterised communication patterns. This contrasts with the conservative system of finite sets used by the previous work. We presented a series of examples illustrating the practical utility and effectiveness of this system, including the control of index calculation in roles, one of the main source of errors in MPI.

The next step in developing this work is the implementation of the model as a library of a mainstream language, where session channels can be implemented as communicators in MPI (e.g. MPI_COMM_WORLD)—communication abstraction for a group of processes. We believe that a communication-safe library of parameterised sessions would increase productivity in parallel algorithms, web-services and other distributed applications. From a theoretical perspective, there are several ways to extend the current system. The index expressions in principals can be extended to richer mathematical operations such as congruence and a more expressive form of exponentiation, preserving the decidability of the type system. More dynamic concepts such as joining and leaving of participants within a session are of interest in web-services and cloud management systems design.

Acknowledgements. I thank Nobuko Yoshida for helpful technical discussions on this material, and Dave Clarke, Iain Phillips, Raymond Hu, Andrew Farell and the anonymous reviewers of Coordination and ICFEM for comments on an earlier version of this paper.

References

1. Alves, S., Fernandez, M., Florido, M., Mackie, I.: Gödel System T revisited. Theoretical Computer Science 411(11-13), 1484–1500 (2010)
2. Amtoft, T., Nielson, F., Nielson, H.R.: Type and behaviour reconstruction for higher-order concurrent programs. J. Funct. Program. 7(3), 321–347 (1997)
3. Amtoft, T., Nielson, H.R., Nielson, F.: Behaviour analysis for validating communication patterns. Software Tools for Technology Transfer 2(1), 13–28 (1998)
4. Ateniese, G., Steiner, M., Tsudik, G.: Authenticated group key agreement and friends. In: CCS, pp. 17–26. ACM, New York (1998)
5. Bejleri, A.: Practical Parameterised Session Types. Tech. Report DTR10-7, Imperial College (2010), http://www.doc.ic.ac.uk/~ab406/papers/param.pdf
6. Bejleri, A., Hu, R., Yoshida, N.: Session-based programming for parallel algorithms: Expressiveness and performance. In: PLACES 2009. EPTCS, vol. 17, pp. 17–29 (2010), http://www.doc.ic.ac.uk/~ab406/parallel_algorithms.html
7. Bettini, L., Coppo, M., D'Antoni, L., Luca, M.D., Dezani-Ciancaglini, M., Yoshida, N.: Global progress in dynamically interleaved multiparty sessions. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 418–433. Springer, Heidelberg (2008)
8. Bonelli, E., Compagnoni, A.: Multipoint session types for a distributed calculus. In: Barthe, G., Fournet, C. (eds.) TGC 2007 and FODO 2008. LNCS, vol. 4912, pp. 240–256. Springer, Heidelberg (2008)
9. Bruni, R., Lanese, I., Melgratti, H., Tuosto, E.: Multiparty Sessions in SOC. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 67–82. Springer, Heidelberg (2008)
10. Bugliesi, M., Castagna, G., Crafa, S.: Access control for mobile agents: The calculus of boxed ambients. ACM Trans. Program. Lang. Syst. 26(1), 57–124 (2004)
11. Caires, L., Vieira, H.T.: Conversation types. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 285–300. Springer, Heidelberg (2009)
12. Carbone, M., Yoshida, N., Honda, K.: Asynchronous session types: Exceptions and multiparty interactions. In: Bernardo, M., Padovani, L., Zavattaro, G. (eds.) SFM 2009. LNCS, vol. 5569, pp. 187–212. Springer, Heidelberg (2009)
13. Castagna, G., Padovani, L.: Contracts for mobile processes. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009 - Concurrency Theory. LNCS, vol. 5710, pp. 211–228. Springer, Heidelberg (2009)
14. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge (1999)
15. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type disciplines for structured communication-based programming. In: Hankin, C. (ed.) ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998)
16. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008, pp. 273–284. ACM, New York (2008)
17. Nelson, N.: Primitive recursive functionals with dependent types. In: Schmidt, D., Main, M.G., Melton, A.C., Mislove, M.W., Brookes, S.D. (eds.) MFPS 1991. LNCS, vol. 598, pp. 125–143. Springer, Heidelberg (1991)
18. Nielson, H.R., Nielson, F.: Higher-order concurrent programs with finite communication topology (extended abstract). In: Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1994, pp. 84–97. ACM, New York (1994)

19. Takeuchi, K., Honda, K., Kubo, M.: An interaction-based language and its typing system. In: Halatsis, C., Philokyprou, G., Maritsas, D., Theodoridis, S. (eds.) PARLE 1994. LNCS, vol. 817, pp. 398–413. Springer, Heidelberg (1994)
20. Web Services Choreography Working Group: Web Services Choreography Description Lang., http://www.w3.org/TR/ws-cdl-10-primer/
21. Yoshida, N., Denielou, P.-M., Bejleri, A., Hu, R.: Parameterised multiparty session types. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 128–145. Springer, Heidelberg (2010)

A Formal Verification Study on the Rotterdam Storm Surge Barrier

Ken Madlener[1], Sjaak Smetsers[1], and Marko van Eekelen[1,2]

[1] Institute for Computing and Information Sciences (iCIS), Radboud University Nijmegen, The Netherlands
[2] School of Computer Science, Open University of the Netherlands
{k.madlener,s.smetsers,m.vaneekelen}@cs.ru.nl

Abstract. This paper presents the results of the validation and verification of a crucial component of BOS, a large safety-critical system that decides when to close and open the Maeslantkering, a storm surge barrier near the city of Rotterdam in the Netherlands. BOS was specified in the formal language Z and model checking has been applied to some of its subsystems during its development. A lightweight model of the C++ code and the Z specification of the component was manually developed in the theorem prover PVS. As a result, some essential mismatches between specification and code were identified. We have also validated the Z specification itself by the use of challenge theorems, to assess particular design choices. Tools have been used to exhaustively search for inconsistencies between the original specification and the challenge theorems, which led to deeper issues with the specification itself.

Keywords: safety-critical systems, lightweight formal methods, PVS, validation of specifications, Z.

1 Introduction

Humans increasingly rely on automation, the advantage being that a computer can make unprejudiced decisions, not being influenced by mood or other conditions. It is therefore often considered to be safer to let a computer be in control. A study showed that this is the case for the Maeslantkering, a storm surge barrier near the Dutch city of Rotterdam. The barrier consists of two hollow floating walls, connected by steel arms to pivot points. Each of these arms is as large as the Eiffel Tower. The barrier operates fully autonomously and is therefore sometimes called the largest robot in the world.

A control system called BOS (Dutch: Beslis & Ondersteunend Systeem) makes the decisions about closing and opening the barrier. This system was developed by Logica Nederland B.V.[1] for Rijkswaterstaat (RWS) - a division of the Dutch Ministry of Transport, Public Works and Water Management. When BOS

[1] Called CMG Den Haag B.V. at the time of the development of BOS.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 287–302, 2010.
© Springer-Verlag Berlin Heidelberg 2010

predicts that the water level will rise so high that it could flood the city of Rotterdam, it has the responsibility to close the barrier. This makes BOS a safety-critical system. On the other hand, Rotterdam is a major port, so the barrier should close only when really necessary. Unnecessarily closing the barrier costs millions of Euros because of restricted ship traffic.

It is often loosely said that even computers make mistakes. Verification and validation efforts must be undertaken to reduce the severity of this risk. The IEC61508 standard [3] recommends the use of formal methods for safety-critical systems. The BOS project adhered to the standard by using the formal language Z [14] in combination with the ZTC type checking tool [4] for specification. Some of its subsystems have been model checked using SPIN [1]. This level of formal support during the development turned out to be a cost-effective approach for the BOS project [15]. The system has been in operation since 1997 and a test closure is performed annually. On November 11th, 2007 BOS closed the barrier on its own for the first time because of a combination of high tide and storm.

With the advent of theorem provers and powerful decision procedures, more rigorous approaches to formal verification come within reach, even for large software systems such as BOS. The Nuclear Research and consultancy Group (NRG) and RWS commissioned Radboud University Nijmegen a 3-month project to do a case-study in applying formal verification to a part of BOS. NRG's field of operation is nuclear applications, where safety standards are even higher. Their objective was to investigate if formal verification can increase confidence in the safety of software. The conducted work is carried out on a crucial component of BOS selected by experts at RWS and Logica of 800 lines of sequential C++ code. The code has been verified against the existing Z specification, and the specification itself has been validated.

Given the man hours available for the actual verification and validation work in the project (roughly 2 months, one PhD student), the task was challenging and presented several hurdles that had to be taken. Since the code had not been formally verified before, there was no strict correspondence between code and specification. To make the two fit together and to isolate the selected component, an understanding of the code that goes deeper than what is written in the formal specification is required. The formalization has been carried out in the PVS theorem prover [9] by means of a manually developed lightweight semantics.

The development of the BOS system is an effort linking several disciplines. This induces the risk of interface problems caused by misunderstandings that undermine robustness. Because a specification is the result of a translation of the designer's intuition into a formal language, it can not be formally verified. We were nevertheless able to semi-systematically validate the specification with the help of challenge theorems. A challenge theorem is a property that from the point of view of the person performing the validation is plausible and should hold. With our understanding of the domain we were able to formulate several challenge theorems. PVS and its testing features enabled us to discover some extreme situations in which the component might behave suspiciously.

The main contribution of this paper is an exposition demonstrating how lightweight modeling of industrial C++ code may enable one to find mismatches between specifications and code, based on a concrete case-study. Although we are confident that our PVS model faithfully reflects the semantics of the source code, future projects that verify safety-critical software may desire formal guarantees. We sketch an approach in this paper to resolve issues regarding the connection between the modeled semantics and the true semantics of the component as future work. This paper also demonstrates how challenge theorems can assist one in validating specifications.

This paper is organized as follows. Section 2 gives a description of the selected component. The model is described in Section 3. Validation of the specifications is discussed in Section 4. The case-study is evaluated in Section 5. Related work is discussed in Section 6 and future work on ensuring correspondence between model and executed code is discussed in Section 7. Section 8 concludes.

2 The Considered Component: DEW

BOS makes decisions based on water level predictions computed from hydrological and meteorological information. When the expected water levels are considered to be too high, then it starts the procedure to close the barrier: commands to a system that operates the valves, pumps and motors of the barrier are sent and authorities are informed (via fax and pager). While in operation, BOS runs a script that continuously makes calls to native functions (written in C++). These functions can be categorized into sending out a command, a status request and a request for a decision. The component we consider in this study, determine excessive water level (DEW), is one of the functions that make the decisions.

A system called SOBEK, developed by Delft Hydraulics[2], generates the water level predictions. It computes model-based predictions for the next 24 hours in steps of 10 minutes for three physical locations: Rotterdam, Dordrecht and Spijkenisse (these are cities in the Netherlands). With each call, DEW receives a number of parameters from the script: for each of the locations a maximum water level and the desired evaluation interval of the predictions. To reduce the load on SOBEK, predictions are stored in a database. DEW obtains the predictions from SOBEK via the hydraulic-model evaluator. The model evaluator first tries to look up the requested prediction in the database, and if it does not exist, issues a new request to SOBEK.

DEW searches the predictions for a point in time where the maximum water level at one of the locations is exceeded by the prediction and it raises a flag if an excess is found. Some threshold is taken into account so that the barrier will only close when the excess is critical. An excess is considered to be *critical* when one of the maximum water levels is exceeded, and 20 minutes later the predicted water level is at least as high. In all other cases DEW will tell the script not to close the barrier.

[2] Now called Deltares.

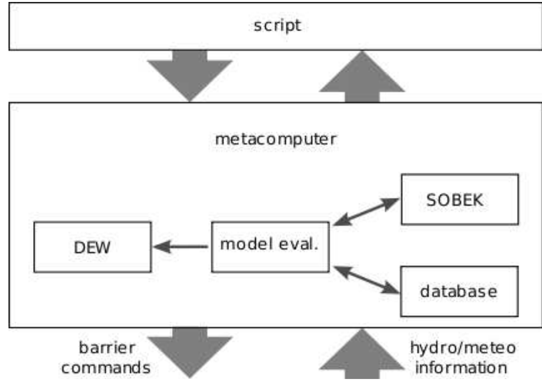


Fig. 1. A schematic representation of the relevant BOS components

Particular about the script language is that it works with lifted values. For example, a boolean in the script can be either `true`, `false` or `undetermined`. This also goes for the three maximum water levels and evaluation period provided by the script.

2.1 Z Specification

The Z specification of DEW, the decision component, is composed of a number of Z schemas using the standard Z operators piping ($\gg$) and disjunction ($\vee$). For presentation purposes we have translated the original Dutch names to English and slightly simplified the formulas. The main schema consists of the following composition:

$$\begin{aligned} DEW == {} & SetEvaluationParams \gg DetModelEvaluation \gg \\ & (EvaluationFailed \vee CoreDEW) \end{aligned}$$

SetEvaluationParams puts the parameters from the script into the appropriate form. The output is passed on using the piping operator to *DetModelEvaluation*, which specifies the behavior of the hydraulic-model evaluator. If *DetModelEvaluation* is successful, then the schema *CoreDEW* is chosen, where the real work of DEW takes place, otherwise, *EvaluationFailed* is chosen.

Schema CoreDEW. For reasons of space we omit some of the Z definitions of the notions introduced below. The schema *CoreDEW* takes two input parameters: *LocList* and *Interval*. *LocList* is a partial function from locations to a maximum water level (it is partial because sometimes not all locations have a determined maximum water level). The prediction data is obtained from an external schema that represents the model evaluator. This is represented as a table of records *LocSeqs* (for predictions sequences per location) in which each record consists of a location l (which corresponds to a location in the domain of *LocList*), a

prediction vals (a function mapping time to the predicted water level), and a point in time TBegin at which the prediction starts. Both time and water levels are represented as natural numbers. The result of $\mathit{CoreDEW}$ consists of a flag Excess indicating whether an excess will occur, and a point in time $\mathit{ExpTime}$ corresponding to the first critical excess. Since it may be the case that such a critical excess does not exist (even if Excess is true), this value is lifted. The condition for the existence of an excess is defined in the design document as:

$$
\begin{aligned}
&\mathit{return}!.\mathit{Excess} = \\
&\quad \textbf{if } (\exists\ s : \mathit{LocSeqs};\ l : \mathrm{dom}\ \mathit{LocList};\ i : \mathbb{N} \bullet \\
&\qquad i \in 1..(\#s.\mathit{vals}) \land s.\mathit{loc} = l \land (s.\mathit{vals}\ i).\mathit{val} > (\mathit{LocList}\ l)) \\
&\quad \textbf{then } \mathit{ExDet\ true} \textbf{ else } \mathit{ExDet\ false}
\end{aligned}
$$

It says that if there is an index i in the domain of $s.\mathit{vals}$ which is greater than the maximum water level at location l given by $\mathit{LocList}\ l$, then there is an excess, otherwise there is no excess.

Critical excesses are excesses such that 20 minutes later the water level is at least as high. (In the BOS documentation these are therefore called *non-decreasing* excesses.) The critical excesses have to lie within the predetermined evaluation interval. The starting time is obtained by taking the minimum of begin times of all locations.

$$
\mathit{TStart} == \min\ \{l : \mathit{LocSeqs}\ \bullet\ l.\mathit{TBegin}\}
$$

For the sake of completeness, we also give the definition of the existence of a critical excess. The details, however, are not important for the rest of this paper.

$$
\begin{aligned}
&\textbf{let } \mathit{ExcessTimes} == \\
&\quad \{s : \mathit{LocSeqs};\ l : \mathrm{dom}\ \mathit{LocList};\ i : \mathbb{N} \mid i \in\ 1..(\#s.\mathit{vals}) - 2 \land \\
&\quad\ s.\mathit{loc} = l \land (s.\mathit{vals}\ i).\mathit{val} > (\mathit{LocList}\ l) \land \\
&\quad\ (s.\mathit{vals}\ (i+2)).\mathit{val} \geq (s.\mathit{vals}\ i).\mathit{val} \bullet \\
&\quad \mathit{TStart} + (i-1) * 10\}\ \cap\ \mathit{Interval} \bullet \\
&(\mathit{ExcessTimes} = \emptyset \Rightarrow \mathit{return}!.\mathit{ExpTime} = \mathit{EtUndet}) \land \\
&(\mathit{ExcessTimes} \neq \emptyset \Rightarrow \mathit{return}!.\mathit{ExpTime} = \mathit{EtDet}(\min\ \mathit{ExcessTimes}))
\end{aligned}
$$

In the above, the indices i of critical excesses are mapped to absolute time (using Z's set comprehension notation). The resulting set is intersected with the evaluation interval and checked to be nonempty. Recall that for an excess to be critical, the water level has to be 20 minutes later at least as high (hence, $i+2$).

A careful reader might have noticed two issues with the above specification. First of all, it might seem suspicious in the first definition the interval is not taken into account. This is a correct observation. Secondly, TStart is chosen to be the first TBegin time of *all* selected prediction sequences. In $\mathit{ExpTime}$, TStart is then used as the same offset for every prediction sequence, even if it has a different TBegin. Both issues have been resolved in the C++ code. In order to be able to prove consistency between Z specification and C++ code, we have fixed these flaws in the Z specification (see Sections 3.2 and 3.3).

3 Formal Analysis

In this section we describe how a model has been created out of the code of DEW and how it was checked and proved to be consistent with the specification. The use of C++ in BOS is according to "safe" coding guidelines (see e.g. [2]); there was no heavy use of object orientation, no pointer arithmetic, etc. This permitted us to develop a lightweight PVS model of the C++ code. The PVS theorem prover was chosen for the very practical reason of locally available expertise. There exist theorem provers for Z such as Z/Eves [11], but our focus is on the development of a lightweight semantics of the component's C++ code and the Z specification could easily be transliterated into PVS. In short, PVS is based on higher-order logic with dependent types and predicate subtyping. We do however not make extensive use of these distinctive features of PVS and we believe that many other theorem provers would also suit the job.

3.1 Translation of C++ to PVS

Datatypes. BOS works with lifted types to represent the possibility of information being undetermined. A number of subclasses are derived from the C++ class `LType` to represent these types. Although PVS has a standard facility to lift types, we model these classes as records to stay close to the original code. The names of lifted types are prefixed by `L` as a naming convention.

```
LType : type = [# fDet : bool #]
LInt : type = LType WITH [# iVal : int #]
```

Setting the value of a lifted integer to 5, i.e. `li.Set(5)`, would be modeled using

```
LInt_Set(i : int): LInt = (# fDet := TRUE, iVal := i #)
```

as `s WITH [somevar := LInt_Set(5)]`, where `s` is the current state. Whenever a set-function is used, the determined-flag is automatically set to `true`. We model C++ integers as unrestricted integers in PVS.

Functions. The C++ functions have been translated into separate PVS theories carrying the same names and taking the same arguments. Arguments that are passed on by reference are modeled by making local PVS variables (using `LET ... IN`) of the function arguments before the beginning of the function body and then updating the returned variables at the end of the function body. For example, the header of the C++ function `CoreDEWC` (corresponding to the Z schema *CoreDEW*)

```
static flag CoreDEWC (
  const LInt []        cLocList,  // in
  const Interval&      cInterval, // in
  flag&                Excess,    // out
  LTime&               ExpTime    // out
)
```

is translated into a PVS function of type

```
CoreDEWC_pvs (
  cLocList: [Loc → LInt],
  cInterval: Interval
) : [flag, flag, LTime]
```

The representation of the type `Interval` closely follows the declaration in C++. This works for functions have no side-effects. The functions were all annotated with in/out flags, so these kind of conversions were straightforward (but, indeed, not formally sound). Each function has been modeled as a separate PVS theory.

In PVS, functions are always total. If PVS is not able to deduce totality by itself, it will generate a proof obligation. This way it is enforced that the execution model terminates, which was not a problem for the considered code of DEW.

For-loop. At the heart of DEW's code lies one for-loop. This for-loop runs through the prediction for a single location and searches for excesses within the specified evaluation interval. It makes use of an object `s` which resembles the Z defined prediction sequence (see Section 2.1). It also assumes that `MaxLevel` contains the maximum water level of location `s.loc` (selected from the input parameter `cLocList`).

```
for(int item = 0; !ExpTime.IsDetermined () &&
  (item < s.GetSize() - 2); ++item) {
  if(TLoop >= TBegin && TLoop <= TEnd) {
    const int next_wl = s.vals.ElementAt(item).val;
    if(next_wl > MaxLevel) {
      Excess = TRUE;
      // is this a critical excess?
      if(s.vals.ElementAt(item + 2).val >= next_wl)
        ExpTime.Set(TLoop);
    }
  }
  TLoop += stepsize;
}
```

`TBegin` and `TEnd` are the boundaries of `Interval`. Note that the outer conditional corresponds with the intersection in the Z definition of critical excesses. Two variables are being set: `Excess`, a boolean which indicates whether an excess occurs in the prediction, and `ExpTime`, the first time at which a critical excess occurs. This for-loop has been modeled as a recursive function. A measure has to be supplied with a recursive function, which tells PVS that the number of iterations left at some point will be 0.

3.2 Communication with Hydraulic-Model Evaluator

To reduce the load on the prediction engine, i.e. SOBEK, previously computed predictions are stored in a database. The hydraulic-model evaluator acts as a

proxy for SOBEK and the database. Calls to the evaluator have to supply the current evaluation time (of the synchronous script). The evaluator then checks whether the prediction already exists in the database, and if it is up to date. If this is the case, it returns the existing prediction. If not, it issues a request to SOBEK to compute a new prediction. The new prediction is then stored in the prediction database with a run-id. The prediction itself is not returned by the evaluator, but only the run-id and a status flag. The request may fail for several reasons: the time of the prediction requested is invalid, SOBEK is in an error state, etc. The status flag `RunStatus` indicates whether the request was successful. If this flag is true, the specification says we may assume that an entry with the run-id exists in the database. We have used this informal information in our PVS model.

From a functional point of view, the model evaluator, SOBEK and the database can be considered as a single entity. Because we do not verify the model evaluator, we treat it as something that has arbitrary output (including its success or failure). The following C++ code obtains the predictions from the database:

```
if(!DB.SelectPrediction(loc, Run.Get()))
{ <...> /* error */ }
else {
  const LocSeq& s = DB.GetLocSeq();
  <...>
}
```

The select operation points `DB` to the right record by changing its internal state. The actual prediction is obtained in the code by `DB.GetLocSeq()` and fed into the for-loop. The code shown above uses the location and run-id (`loc`, `Run.Get()`) as a key to return a unique prediction. In the Z specification, the database may contain several predictions starting at various times. To avoid obvious inconsistencies, we have chosen to change the Z specification so that it also contains a unique prediction per key. This modification was made in accordance with a system expert.

To simulate all possible outputs of the model-evaluator (and thereby also SOBEK and the database), we use uninterpreted variables to represent success/failure of the model evaluator and the database output. The correctness proof has to take every possible value of the variables in account.

```
RunStatus : bool
Seq : type = [# n: Length, vals: [below(n) → height],
                TBegin: Time #]
LocSeqs : [[Loc, nat] → Seq]
```

The model evaluator is started once per execution of DEW, so we let `RunStatus` to be a constant. The C++ functions `SelectPrediction` and `GetLocSeq` are represented in PVS by the the following functions:

```
DB_SelectPrediction : bool = RunStatus
DB_GetLocSeq(loc: Loc, RunId: nat) : Seq = LocSeqs(loc, RunId)
```

We do not have an explicit state of `DB` in the model, so we have to supply `loc` and `RunId` as arguments to `DB_GetLocSeq`. With this way of modeling we would have to supply the parameters that initialize a particular object with every function call that is made on it. This is not a useful solution for code which uses many objects with internal states, but works well for the code of DEW.

3.3 Verification

The result of a formal verification is either one or more discrepancies between specification and code or a proof that (model of) the code implements the specified behavior. It is common that most discrepancies (if there are any) are already found during the process of modeling. While modeling DEW we found two discrepancies, and we found a third during formal verification itself.

The first problem that was found as a result of modeling is the following. For regular excesses, the evaluation interval was not being taken into account in the specification (see Section 2.1), but was in the implementation. This is a design decision of the implementer which happened to be correct. However, the fact that the specification was not updated accordingly, suggests that the implementer was not aware that he was actually fixing something. The second problem found during modeling had to do with the prediction database. In the Z specification, the database may store multiple predictions per location and run-id. In the C++ code, only one prediction per location/run-id is considered (which is obtained from the database, see Section 3.2). An assumption that we were not aware of might resolve this issue, but this was not obvious from the specification nor its guiding text.

The main part of the formal verification itself requires proof of the following lemma that equates the specification and implementation (as functions):

```
correctness : LEMMA
    ∀ (param: CoreDEWIn):
        CoreDEWZ_pvs(param) = CoreDEWC_pvs(param)
```

Both return a triple `[flag, flag, LTime]` (the return flag, `Excess`, `ExpTime`, resp.) which all have to agree under `P` and every possible `param`.

Proof approach. Instead of directly trying to prove the above lemma in PVS, we chose to first develop a toy model in the model checker SPIN [1] to experiment with. The reason is that when developing non-trivial formalizations in a theorem prover, often a considerable amount of time is devoted to debugging specifications and theorems. Using SPIN, we could try out candidate invariants by putting `assert` commands in the model code. The number of possible inputs was incremented until enough confidence in the correctness of the model was obtained. This essentially comprises playing with the ranges of `length`, `height` and the maximum water levels. While doing so, we found a flaw in the code: the last two elements of the array containing the predictions are not being taken into account. After fixing this issue, no other issues were found and the theorem could be proved in the first attempt with approx. 2000 proof commands.

4 Validation of the Specification

Interaction of components that are developed by engineers of several disciplines poses the risk of problems with interfacing as a result of misunderstandings. The standard way to validate a specification is picking different examples of system behavior allowed by that particular specification, and see if they fit with the intuition of the designer. This is often infeasible to do exhaustively, hence, this method is sound, but not necessarily complete.

To assess the design choices made in the specification in a more systematic way, we have formulated *challenge theorems*. A challenge theorem is a statement that from the point of view of the person performing the validation might be a valid consequence of the specification. By checking the consistency of a challenge theorem with the existing specification, which may be done automatically with the help of tools, counterexamples are generated that demonstrate how the existing specification and the challenge theorem diverge. These examples may serve as concrete material for discussion with the experts. Accordingly, the existing specification has to be altered or the domain understanding of the person performing the validation has to be improved.

Stating the most general system property, and focusing it gradually on DEW forces us to identify what assumptions we make about other components. For closing the barrier, these are based on the following general theorems:

– If there will be a flood, then the barrier will be closed in a timely manner.
– If the barrier is closed, there would have otherwise been a flood.

For DEW in particular, this means that we have to assume that the predictions are correct and if it decides to close the barrier, then this is properly delegated by BOS as a command to the barrier. The theorems for DEW become:

– If there is an excess in the available predictions that is critical, then DEW will decide to close.
– If DEW has decided to close, there would otherwise have been a critical excess.

We have verified in Section 3 that w.r.t. the existing definition of critical excess that DEW behaves correctly. In Section 4.2 we validate the definition of critical excess itself.

4.1 Decision Based on Incomplete Information

We have for the moment ignored the possibility of failing sensors. This makes the actual case a bit more complicated, because if no prediction data is available at all or is only partially available, then DEW can simply not make a sensible decision and should (and does) therefore raise an alarm. However, when the predictions are only partially available, it should still decide to close the barrier if needed. With each call to DEW, for each of the three locations a maximum water level is supplied which may be undetermined. If one of these locations is

undetermined, and no excesses were found on the determined locations, then DEW would say there is no excess. Both Z and C++ agree in this, but this behavior is obviously not safe. The experts agreed with us that this is an issue, but the way DEW is called precludes this problem.

Another issue we found is that the evaluation period (chosen by the script) may not necessarily be within the period of which the predictions are available. In this case DEW would look at the intersection of the two (see the Z specification for critical excesses in Section 2.1), which again is not safe.

4.2 Critical Excesses

In order to assess the DEW's specified conditions to close the barrier, we have formulated two alternative definitions of a critical excess. These variations were motivated by real-life, but possibly rare, scenarios. Without loss of generality we focus on predictions for one location, ℓ say.

As an extreme example, consider a tsunami. In the predictions this would typically look like a short but high peak. Based on this idea, we have defined our own criterion in (1) that expresses that the "volume" (to be loosely interpreted) exceeding the maximum water level may not surpass a predefined amount M (here P_ℓ and max_ℓ denote the prediction and maximum water level respectively):

$$\{i : \mathbb{N} \mid \sum_{k=i}^{\#\mathrm{dom}\ \mathrm{P}_\ell} \max(0, \mathrm{P}_\ell(k) - \mathrm{max}_\ell) > M\} \tag{1}$$

Another situation one typically wants to avoid is a possibly slight, but continuous excess of the maximum water level:

$$\{i : 1 \ldots \#\mathrm{dom}\ \mathrm{P}_\ell - T \mid \forall j \in 1 \ldots T :\ \mathrm{P}_\ell(i + j) > \mathrm{max}_\ell\} \tag{2}$$

There are obvious cases in which these two variants do not coincide. In (3) below, the original definition for critical excess as in the Z specification (see Section 2.1) is expressed:

$$\{i : \mathbb{N} \mid i \in 1 \ldots \mathrm{length}_\ell - 2\ \wedge\ \mathrm{P}_\ell(i + 2) \geq \mathrm{P}_\ell(i) > \mathrm{max}_\ell\} \tag{3}$$

All three versions have been entered into PVS and lemmas (although unprovable) that express their equivalence were formulated. These lemmas implicitly quantify over the PVS variable `LocSeqs` (see Section 3.2) and the maximum water level. We used a random testing feature [8] in PVS that allows the user to audit the truth of simple lemmas by trying out a number of instances.

Comparing (1), (2) and (3) gave us an example in the lines of Figure 2, where the water level increases rapidly and then steadily decreases until it is below the maximum water level (indicated by the dotted line). To search for different classes of counterexamples, we added the premise that a rapid increase followed by a slow decrease until maximum water level does not occur in the predictions of `LocSeqs`. This resulted in another type of examples, where the water level first

increases rapidly, drops and then increases again. It turned out that in those cases the barriers would close too late, as shown in Figure 3. It was not clear whether these scenarios may occur in reality or are precluded by undocumented assumptions about behavior of water, so experts were consulted, see Section 5.

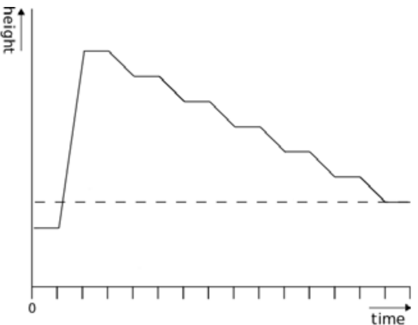


Fig. 2. No non-decreasing excess

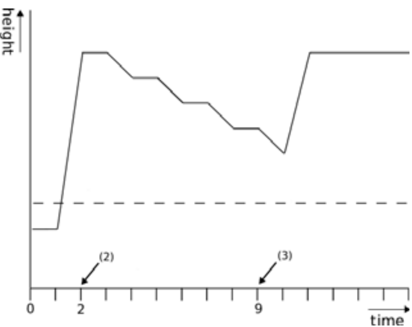


Fig. 3. At $t = 2$ the barrier closes according to (2), at $t = 9$ it closes according to (3), the Z-spec

5 Case-Study Evaluation

Verification. The original BOS project applied formal methods in the form of formal specification in Z to discover bugs in an early stage of the development. The specifications were used as a reference for implementation. The code is a composition of Z schemas translated into blocks of C++ annotated with the corresponding schema name. The implementation of each schema is based on the programmer's own interpretations and design decisions. Hence there is no real formal connection between C++ implementation and Z specification. We have found three inconsistencies as a result of this. Two of them became apparent during the formalization in PVS, but a third was missed until real verification was applied. In the first two cases, the code seems to fix a flaw in the specification. Whether these fixes are a deliberate decision or just "luck" is unclear. The manual modeling of C++ code in PVS provided enough assurance about the correspondence between source code and model for this particular case-study. In Section 7 we discuss a complementary approach that may be viable for future verification projects on BOS or other safety-critical software.

Validation. We have validated the specification of DEW itself by formulating challenge theorems that focus on a particular high-level property. We considered two aspects: safety and the condition under which DEW decides to close the barrier. Validation of the safety aspect showed that there are two situations in which DEW does not raise an alarm when information is missing (due to undetermined values or incomplete predictions). To validate the closing conditions,

we studied the definition of critical excess by comparing it with plausible alternative definitions. In principle these definitions do not have to be complete or correct, as the resulting counterexamples are only used as a guidance to understand the differences between the definitions and for discussion with subject matter experts. Possibly, several iterations are required to synchronize with the domain experts in other projects. We believe that in this case-study our unprejudiced abstract understanding of the domain was an advantage in finding the issues with the specification.

Impact analysis. Based on our findings presented in a preliminary report, subject matter experts performed an impact analysis. They agreed that the issues we found are possible and that the system would exhibit undesirable behavior in those situations. However, the issues found are very unlikely to do harm in reality, because the delays caused are very small compared to the process of detecting a storm and preparing for a closure. So the findings have no impact on the performance of BOS required in practice.

6 Related Work

BOS was developed using what one could consider to be lightweight formal methods. It was specified in Z to discover ambiguities in an early stage of its development and parts of it were checked using the SPIN model checker [1]. Experiences with the development of BOS using this approach are described in [16]. In [15], Hall's seven myths of formal methods are revisited based on the experiences of the BOS project.

The interface between BOS and BESW (a separate system that operates the barrier) was studied initially by Kars [5]. The interface was specified in Promela and model checking was applied using SPIN [1], revealing several flaws. A redesigned and redefined version was developed by RWS and subsequently studied by Ruys [10] again using Promela and SPIN, applying several abstraction techniques to fight state space explosion. One serious timing error was found.

The emergency closing system of a different storm surge barrier, located in the Eastern Scheld, has been a case-study of the QUEST project in the late 1990ies, which was funded by the German BSI [13]. This project focused on the combination of formal methods in the software development process. One of the aims was a formal verification of the correctness of the open- and close-signals of the barrier.

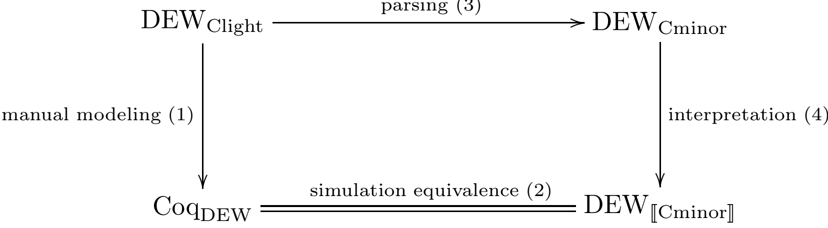
The approach of a lightweight modeling of the source language semantics way has been practiced before in [12] to verify Fiasco's IPC implementation. John et al. have developed a tool that generates out of MISRA-C state transition models encoded in PVS. Due to the fact that PVS is strongly typed, possible run-time errors in the C program result in inconsistencies in the generated PVS specification.

7 Future Work: Certified Lightweight Semantics

Industrial safety-critical software, like the decision and support system (BOS) studied in this paper, is usually developed according to "safe" coding guidelines, e.g. see [2]. These coding rules attempt to increase the ability to more thoroughly check the reliability of such critical applications. In essence, the rules restrict the use of the complete source language to a subset of that language. The advantage of this subset is that its semantics if often much simpler than the full semantics of the complete language. For example, the DEW component of BOS is programmed in C++, but hardly uses any of the object oriented features available in C++. In fact, this made the simple direct translation to PVS (Section 3) feasible.

This PVS model, however, was created manually, and may therefore not completely match the original semantics of DEW. For our project the informal justification of correctness of this model was sufficient, but for future verification projects a stronger, preferably provable correspondence between the original code and the derived model might be desired. In the section we discuss how such a correspondence can be established.

Formalizing semantics of real programming languages is a very important trend in computer science. One of the greatest achievements in this area is a fully verified optimizing C-compiler developed in the CompCert project [7]. In this project a large subset of C, called Clight, is compiled to PowerPC code. The correctness of this compiler, containing various complex optimization passes, has been proven in the Coq theorem prover. By ignoring the small differences between Clight and the subset of C++ in which DEW is specified, we can consider DEW as a Clight program. The semantics of Clight is defined by means of an interpreter for Cminor which is, in fact, a representation in Coq of the abstract syntax tree created by the Clight parser. This framework can then be used to obtain what we brand as the "official" semantics of the DEW code. On the other hand, we can use the manually derived model presented in this paper (but now specified in Coq rather than in PVS) as an alternative semantics, and show correctness of this model by proving in Coq that both semantics are equivalent. This proof actually boils down to constructing the simulation relation as depicted in the following diagram.



In general, to prove a property of a Clight program, say P, there are two possibilities:

1. By manually constructing a model (step 1 in the diagram) for P, proving the desired property using this model, and showing the equivalence of the model (step 2) based on the official semantics of P (which is obtained via step 3 and 4 in the diagram).
2. By using the official semantics of P directly, i.e. both formulate and prove the desired property directly in $\mathrm{DEW}_{[\![\mathrm{Cminor}]\!]}$

The main advantage of the first approach is that it provides a *lightweight shallow embedding*: The program is directly translated into Coq in such a way that only those aspects of P that are relevant to the desired property are take into account. Proving properties using such a lightweight abstract model is usually much easier than to use a full concrete model (a model incorporating all the aspects of the entire language). The disadvantage is, of course, that it requires the additional proof obligation in which correspondence between model and official semantics is stated.

Last but certainly not least, by compiling our program with CompCert, we obtain for free that resulting code exhibits exactly the same behavior as the model about which the safety properties have been proved: the verified compiler guarantees that properties proved on the (model of the) source code hold for the executable compiled code as well.

8 Conclusions

In this paper we have verified and validated a crucial component of BOS, called DEW. This component is responsible for deciding when the storm surge barrier near Rotterdam should close. BOS was specified in the formal language Z, but no real formal verification was applied during its development. We have developed a formal model of the Z specification and the C++ code in the PVS theorem prover by means of a manual lightweight modeling of the semantics. This enabled us to find an error in the code, but also two flaws in the Z specification which the code seems to fix. The challenge of verifying DEW was to make a sound model that isolates the relevant code.

Validation of the specification itself revealed deeper issues with the specified and implemented behavior of DEW. It does not raise an alarm when information is missing that is mandatory for the decision being made, which is questionable behavior in the context in which it is being used. Another issue is that the precise conditions under which DEW decides that the barrier must close seem to be incomplete. This last issue seemed quite serious. All issues were confirmed by domain experts.

In future work we plan to carry out the proposed approach to ensure (formal) correspondence between lightweight manual modeling and official semantics on other safety-critical software.

Acknowledgements. We thank Wouter Geurts from Logica Nederland B.V. for the technical support during the project. We also thank Erik Poll and the anonymous reviewers for their valuable comments on a draft version of this paper.

References

1. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
2. Holzmann, G.J.: The power of 10: Rules for developing safety-critical code. IEEE Computer 39(6), 95–97 (2006)
3. IEC: Functional safety: Safety related systems, International Standard IEC 61508, International Electrotechnical Commission, Geneva, Switzerland (1996)
4. Jia, X.: ZTC: A Type Checker for Z – User's Guide. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, USA (1994)
5. Kars, P.: The application of Promela and SPIN in the BOS project. In: Grégoire, J.-C., Holzmann, G.J., Peled, D.A. (eds.) Proc. of SPIN 1996: The Second Workshop on the SPIN Verification System. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 32. Rutgers University, New Jersey, USA (1996)
6. Kars, P.: Formal methods in the design of a storm surge barrier control system. In: Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems (1996)
7. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
8. Owre, S.: Random testing in PVS. In: Workshop on Automated Formal Methods, Seattle, USA (2006), http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf
9. Owre, S., Rushby, J.M., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
10. Ruys, T.: Towards Effective Model Checking. PhD thesis, University of Twente (2001)
11. Saaltink, M.: The Z/Eves system. In: Till, D., Bowen, J.P., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–88. Springer, Heidelberg (1997)
12. Schierboom, E., Tamalet, A., Tews, H., van Eekelen, M., Smetsers, S.: Preemption abstraction: A lightweight approach to modelling concurrency. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 149–164. Springer, Heidelberg (2009)
13. Slotosch, O.: Overview over the project Quest. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 346–350. Springer, Heidelberg (1999)
14. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall International Series in Computer Science (1989)
15. Tretmans, J., Wijbrans, K., Chaudron, M.: Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods. Formal Methods in System Design 19(2), 195–215 (2001)
16. Wijbrans, K., Buve, F., Rijkers, R., Geurts, W.: Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 419–424. Springer, Heidelberg (2008)

Formalization and Correctness of the PALS Architectural Pattern for Distributed Real-Time Systems

José Meseguer[1] and Peter Csaba Ölveczky[2]

[1] University of Illinois at Urbana-Champaign
[2] University of Oslo

Abstract. Many Distributed Real-Time Systems (DRTS), such as integrated modular avionics systems and distributed control systems in motor vehicles, are made up of a collection of components that communicate asynchronously and that must change their state and respond to environment inputs within hard real-time bounds. Such systems are often safety-critical and need to be certified; but their certification is currently very hard due to their distributed nature. The *Physically Asynchronous Logically Synchronous* (PALS) architectural pattern can greatly reduce the design and verification complexities of achieving virtual synchrony in a DRTS. This work presents a formal specification of PALS as a formal model transformation that maps a synchronous design, together with performance bounds of the underlying infrastructure, to a formal DRTS specification that is semantically equivalent to the synchronous design. This semantic equivalence is proved, showing that the formal verification of temporal logic properties of the DRTS can be reduced to their verification on the much simpler synchronous design. An avionics system case study illustrates the usefulness of PALS for formal verification purposes.

1 Introduction

Many Distributed Real-Time Systems (DRTS), such as integrated modular avionics systems and distributed control systems in motor vehicles, are made up of a collection of components that communicate asynchronously and that must change state and respond to environment inputs within hard real-time bounds. Because of physical and fault tolerance requirements, such systems are asynchronous, with each component having its own local clock. Yet, overall system behavior must ensure *virtual synchrony*, in the sense that each cycle of interaction of each system component with the environment and with the other components should result in a proper state change and proper outputs being produced at each component within hard real-time bounds. That is, the system, although asynchronous, must behave *as if it were synchronous*, not in some fictional logical time, but *in actual physical time*.

The design, verification, and implementation of such systems is a challenging and error-prone task for several reasons. The main danger is for a DRTS of this

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 303–320, 2010.
© Springer-Verlag Berlin Heidelberg 2010

nature to enter an *inconsistent state* due to race conditions, network delays, and clock skews in the asynchronous communication between components that can easily fool one component into mistakenly acting on inputs from the wrong cycle, or sending its outputs to other components at the wrong time; that is, the intrinsically asynchronous nature of the system makes it hard to ensure its virtual synchrony. Furthermore, since such a system is often safety-critical, its must undergo a stringent certification process that requires full coverage of the verification of its design and the validation of its implementation. Such a certification effort can be very demanding and time consuming because the state space explosion caused by the system's concurrency can easily make it unfeasible to apply automatic model checking techniques to verify that its design satisfies the required safety properties.

A useful way to meet engineering challenges such as the one described above is to amortize the use of formal methods not on an individual design, but on a *generic family of system designs* by means of a *formal architectural pattern*, that is, a generic formal specification of an engineering solution to a generic design problem that: (i) is shown to be correct by construction; (ii) comes with strong formal guarantees; and (iii) greatly reduces system complexity, making system verification and correct system implementation orders of magnitude simpler than if the pattern were not used. In this paper we present a formal specification and a proof of correctness for one such pattern, namely the *Physically Asynchronous Logically Synchronous* (PALS) architectural pattern, which we have developed in collaboration with colleagues at Rockwell-Collins and UIUC (see [10,1,14]). This pattern provides a generic engineering solution to the problem of designing a DRTS that must be virtually synchronous in spite of its asynchronous nature.

The PALS Formal Model in a Nutshell. The key idea of PALS is to drastically reduce the effort of designing, verifying, and implementing a DRTS of this kind by *reducing its design and verification to that of its much simpler synchronous version*. This is achieved by assuming that the DRTS can rely on an underlying Asynchronous Bounded Delay (ABD) Network (see [3,16]) infrastructure, so that a bound can be given for the delay of any message transmission from any process to any other process. Similarly, it is assumed that the *clock skew* between the different local clocks of the DRTS is bounded. The PALS pattern can then be formalized as a *model transformation* which sends a synchronous system design to its correct-by-construction asynchronous design. Specifically, PALS is a formal model transformation of the form: $(\mathcal{E}, \Gamma) \mapsto \mathcal{A}(\mathcal{E}, \Gamma)$, where:

1. $\mathcal{E}$ is a synchronous system, which is formally defined as an *ensemble of state machines* connected together by a *wiring diagram*.
2. Γ specifies the following *performance bounds*: (i) the *clock skew* of the local asynchronous clocks for each state machine is strictly smaller than ϵ; (ii) the minimum and maximum duration times $0 \leq \alpha_{min} \leq \alpha_{max}$ for any machine to consume inputs, make a transition, and produce outputs; and (iii) the minimum and maximum *message transmission delays* $0 \leq \mu_{min} \leq \mu_{max}$ for communication between any two processes in the ABD network,

and where $\mathcal{A}(\mathcal{E}, \Gamma)$ then denotes the corresponding asynchronous design guaranteed to behave like $\mathcal{E}$ in a virtually synchronous way under the assumption that the performance bounds Γ are met by the underlying infrastructure. As we further discuss below, a key advantage of PALS for formal verification purposes is that the, typically unfeasible, verification of formal requirements for $\mathcal{A}(\mathcal{E}, \Gamma)$ can be reduced to the much simpler verification of such requirements for $\mathcal{E}$.

Main Contributions. This work complements other research on PALS such as [10,1,14] by providing both a formal specification of the PALS architecture and a detailed proof of its correctness that justifies why a formal verification of the synchronous design also verifies its PALS asynchronous version. Specifically, it presents the following contributions:

1. A formal model in rewriting logic [8] of the PALS transformation, expressed in the Real-Time Maude formal specification language [11], including precise requirements for the allowable synchronous designs to which PALS can be applied and the real-time bounds of the infrastructure.
2. A precise derivation of the (smallest possible) period of the asynchronous design $\mathcal{A}(\mathcal{E}, \Gamma)$, and a proof of its *optimality* under the given assumptions.
3. A mathematical justification of a method that *reduces* the formal verification of temporal logic safety and liveness properties of an asynchronous PALS design to the model checking verification of its synchronous counterpart.
4. An avionics case study illustrating the usefulness of the PALS pattern.

The rest of this paper is organized as follows. Section 2 summarizes the basic prerequisites about Real-Time Maude needed to define PALS. Section 3 gives a formal definition of the synchronous models that are inputs for the PALS transformation. Section 4 defines the assumptions about clock drift, network delays, and machine execution times that are also inputs to the PALS transformation, and gives a brief overview of PALS. Section 5 gives a formal specification in Real-Time Maude of the resulting PALS-transformed asynchronous system. Section 6 gives a theorem that makes explicit the temporal logic properties that would have to be verified in the asynchronous model $\mathcal{A}(\mathcal{E}, \Gamma)$ but are reduced to the verification of corresponding properties in $\mathcal{E}$. Section 7 shows the benefit of PALS on an avionics system. Related work is discussed in Section 8, and some final conclusions are drawn in Section 9.

2 Real-Time Maude

A Real-Time Maude [11] *timed module* specifies a *real-time rewrite theory* of the form (Σ, E, IR, TR), where:

– (Σ, E) is a *membership equational logic* [4] theory with Σ a signature[1] and E a set of *confluent and terminating conditional equations*. (Σ, E) specifies the system's states as an algebraic data type, and must contain a specification of a sort `Time` modeling the (discrete or dense) time domain.

[1] I.e., Σ is a set of declarations of *sorts*, *subsorts*, and *function symbols*.

- *IR* is a set of (possibly conditional) *labeled instantaneous rewrite rules* specifying the system's *instantaneous* (i.e., zero-time) local transitions, written with syntax `rl [`l`] : `u` => `v, where l is a *label*. Such a rule specifies a *one-step transition* from an instance of term u to the corresponding instance of term v. The rules are applied *modulo* the equations E.[2]
- *TR* is a set of (usually conditional) *tick rewrite rules*, written with syntax `crl [`l`] : {`u`} => {`v`} in time `τ` if `*cond*, that model time elapse. `{_}` is a built-in constructor of sort `GlobalSystem`, and τ is a term of sort `Time` that denotes the *duration* of the rewrite.

The initial state must be a ground term of sort `GlobalSystem` and must be reducible to a term of the form `{`u`}` using the equations in the specification.

The Real-Time Maude syntax is fairly intuitive. For example, a function symbol f in Σ is declared with the syntax `op` f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Equations are written with syntax `eq` u `=` v, and `ceq` u `=` v `if` *cond* for conditional equations. The mathematical variables in such statements are declared with the keywords `var` and `vars` (see [4]).

In object-oriented Real-Time Maude modules, a *class* declaration

`class` C `|` att_1 `:` s_1`, ... ,` att_n `:` s_n `.`

declares a class C with attributes att_1 to att_n of sorts s_1 to s_n. An *object* of class C is represented as a term `<` $O : C \mid att_1 : val_1, ..., att_n : val_n$ `>` where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. In an object-oriented system, the state is a term of sort `Configuration`, and has the structure of a *multiset* of objects and messages, with multiset union denoted by a juxtaposition operator that is declared associative and commutative, so that rewriting is *multiset rewriting* supported in Real-Time Maude.

The dynamic behavior of concurrent object systems is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule

```
rl [l] : m(O,w)  < O : C | a1 : x, a2 : O', a3 : z >   =>
                 < O : C | a1 : x + w, a2 : O', a3 : z >  dly(m'(O'),x) .
```

defines a parametrized family of transitions in which a message `m`, with parameters `O` and `w`, is read and consumed by an object `O` of class `C`. The transitions change the attribute `a1` of the object `O` and send a new message `m'(O')` *with delay* `x`. "Irrelevant" attributes (such as `a3`) need not be mentioned in a rule.

A *subclass* inherits all the attributes and rules of its superclasses.

A Real-Time Maude specification is *executable*, and the tool offers a variety of formal analysis methods. The *rewrite* command simulates *one* fair behavior of the system, starting with a given initial state, *up to a certain duration*. The

[2] E is a union $E' \cup A$, where A is a set of equational axioms such as associativity, commutativity, and identity, so that deduction is performed *modulo* A. Operationally, a term is reduced to its E'-normal form modulo A before any rewrite rule is applied.

search command uses a breadth-first strategy to analyze all possible behaviors of the system from an initial state, by checking whether a state matching a *pattern* and satisfying a *condition* can be reached from the initial state. Real-Time Maude also extends Maude's *linear temporal logic model checker* to check whether each behavior from an initial state, possibly up to a time bound, satisfies a linear temporal logic formula.

3 Formal Definition of the Synchronous Model

The synchronous model of computation is a synchronous composition of a collection of *nondeterministic typed machines*, an *environment*, and a *wiring diagram* that connects the machines.

Definition 1. *A* (nondeterministic) typed machine $M = (D_i, S, D_o, \delta_M)$ *consists of:*

- D_i*, called the* input set*, a nonempty set of the form* $D_i = D_{i_1} \times \cdots \times D_{i_n}$*, for* $n \geq 1$*, where* $D_{i_1}, \ldots, D_{i_n}$ *are called the* input data types*.*
- S*, a nonempty set, called the* set of states*.*
- D_o*, called the* output set*, a nonempty set of the form* $D_o = D_{o_1} \times \cdots \times D_{o_m}$*, for* $m \geq 1$*, where* $D_{o_1}, \ldots, D_{o_m}$ *are called the* output data types*.*
- δ_M*, called the* transition relation*, a* total *relation* $\delta_M \subseteq (D_i \times S) \times (S \times D_o)$*.*

M *is* finite *iff* D_i*,* S*, and* D_o *are all finite.* M *is* deterministic *iff* δ_M *is a function.*

That is, a machine has n input ports and m output ports; an input to port k should be an element of D_{i_k}, and an output from port j should be an element of D_{o_j}.

Typed machines can be "wired together" into arbitrary sequential and parallel compositions by means of a "wiring diagram," as the one shown in Fig. 1, where the types are left implicit, but where it is assumed that the type in an output wire must match any types in the input wires connected with it.

Definition 2. *A (typed)* machine ensemble $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$ *has:*

- J*, a nonempty finite set of* indices*, and* $e \notin J$ *the* environment index*.*
- $\{M_j\}_{j \in J}$*, a* J*-indexed family of typed machines.*
- E*, called a* typed environment*, is an ordered pair of sets* $E = (D^e_i, D^e_o)$*, where* D^e_i*, the environment's* input set *(inputs* to *the environment), is a nonempty set of the form* $D^e_i = D^e_{i_1} \times \cdots \times D^e_{i_{n_e}}$*, for* $n_e \geq 1$*, and* D^e_o*, the environment's* output set*, is a nonempty set of the form* $D^e_o = D^e_{o_1} \times \cdots \times D^e_{o_{m_e}}$*, for* $m_e \geq 1$*.*
- src*, a function that assigns to each input port* (j, n) *(the input port number* n *of machine* j*) the corresponding output port (or "source" for that input)* $src(j, n)$*. Formally, we define the set of input ports and output ports as follows:*

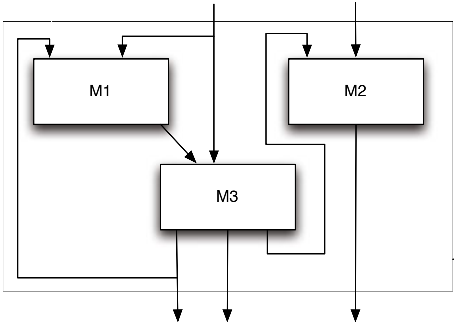


Fig. 1. A machine ensemble

- $In_{\mathcal{E}} = \{(j,n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq n_j\}$, *where* M_j *has* n_j *inputs*
- $Out_{\mathcal{E}} = \{(j,n) \in (J \cup \{e\}) \times \mathbb{N} \mid 1 \leq n \leq m_j\}$, *where* M_j *has* m_j *outputs.*

Then src is a surjective function $src : In_{\mathcal{E}} \to Out_{\mathcal{E}}$ *assigning to each input port the output port to which it is connected, and such that "the types match". In addition, we require that there are no self-loops from the environment to itself; that is, for* $(e,q) \in In_{\mathcal{E}}$, *if* $src(e,q) = (k,p)$, *then* $k \in J$.

An ensemble $\mathcal{E}$ has a *lock-step synchronous semantics*, in the sense that the transitions of all the machines are performed simultaneously, and whenever a machine has a feedback wire to itself and/or to any other machine, then the corresponding output becomes an input for any such machine at the *next* instant. As explained below, what this means is that any ensemble $\mathcal{E}$ is semantically equivalent to a *single state machine*, called the *synchronous composition* of all the machines in the ensemble $\mathcal{E}$. For example, in Fig. 1, the synchronous composition of the typed machines M_1, M_2, and M_3 can be seen as the single machine denoted by the outer box, which hides the internal details of how the machine ensemble is decomposed.

Definition 3. *Given a machine ensemble* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j \in J}, E, src)$, *its* synchronous composition *is the typed machine* $M_{\mathcal{E}} = (D_i^{\mathcal{E}}, S^{\mathcal{E}}, D_o^{\mathcal{E}}, \delta_{\mathcal{E}})$ *where*

- $D_i^{\mathcal{E}} = D_o^e$ *(the input set of the composition is the output set of the environment)*
- $D_o^{\mathcal{E}} = D_i^e$ *(the output set is the input set of the environment)*
- $S^{\mathcal{E}} = (\Pi_{j \in J} S_j) \times (\Pi_{j \in J} D_{OF}^j)$, *where if* $D_o^j = D_{o_1}^j \times \cdots \times D_{o_{m_j}}^j$ *is the output set of* M_j, *then* D_{OF}^j *is the set* $D_{OF}^j = D_{OF_1}^j \times \cdots \times D_{OF_{m_j}}^j$, *where, for* $1 \leq m \leq m_j$, $D_{OF_m}^j = D_{o_m}^j$ *if* $(j,m) = src(l,q)$ *for some* $l \in J$, *and* $D_{OF_m}^j = 1$ *otherwise, with* $1 = \{*\}$ *a one point set. Intuitively,* D_{OF}^j *stores the "feedback outputs" of machine* M_j. *We then have a* "feedback output" *function* $fout_j : D_o^j \to D_{OF}^j$, *where for* $1 \leq m \leq m_j$, *we have* $\pi_m(fout_j(d_1, \ldots, d_{m_j})) = d_m$ *if* $(j,m) = src(l,q)$ *for some* $l \in J$, *and* $\pi_m(fout_j(d_1, \ldots, d_{m_j})) = *$ *otherwise,*

with π_m *the m-th projection from the Cartesian product* D^j_{OF}. *Similarly, for each* $k \in J$ *we have an obvious* input *function* $in_k : D^e_o \times \Pi_{j\in J} D^j_{OF} \to D^k_i$, *where for* $1 \le n \le n_k$, *with* $src(k, n) = (l, q)$, *we have* $\pi_n(in_k(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j\in J})) =$ *if* $l = e$ *then* $\pi_q(\boldsymbol{d})$ *else* $\pi_q(\boldsymbol{d_l})$ *fi, where* π_q *denotes the q-th projection from the corresponding Cartesian product.*

- *The transition relation for* $M_{\mathcal{E}}$ *is the relation* $\delta_{\mathcal{E}} \subseteq (D^{\mathcal{E}}_i \times S^{\mathcal{E}}) \times (S^{\mathcal{E}} \times D^{\mathcal{E}}_o)$, *where* $((\boldsymbol{d}, (\{s_j\}_{j\in J}, \{\boldsymbol{d_j}\}_{j\in J})), ((\{s'_j\}_{j\in J}, \{\boldsymbol{d'_j}\}), \boldsymbol{d'})) \in \delta_{\mathcal{E}}$ *iff, for each* $l \in J$, *there exists* $(s'_l, \boldsymbol{d''_l})$ *such that* $((in_l(\boldsymbol{d}, \{\boldsymbol{d_j}\}_{j\in J}), s_l), (s'_l, \boldsymbol{d''_l})) \in \delta_{M_l}$, *and where* $\boldsymbol{d'_l} = fout_l(\boldsymbol{d''_l})$ *and the output to the environment* $\boldsymbol{d'}$ *is defined for each* $1 \le n \le n_e$ *with* $src(e, n) = (j', r)$ *by* $\pi_n(\boldsymbol{d'}) = \pi_r(\boldsymbol{d''_{j'}})$. *Note that* $\delta_{\mathcal{E}}$ *is a total relation, since each* δ_{M_l} *is a total relation; therefore, some desired* $(s'_l, \boldsymbol{d''_l})$ *always exists. Furthermore, if each machine* M_i *is a deterministic typed machine, then* $M_{\mathcal{E}}$ *is also a deterministic typed machine.*

We assume an environment where the *constraints on the values generated by the environment* can be defined as a *satisfiable* predicate $c_e : D^e_o \to Bool$ so that $c_e(d^e_1, \ldots, d^e_{o_{m_e}})$ is *true* if and only if the environment can generate output $(d^e_1, \ldots, d^e_{o_{m_e}})$. We can associate a transition system defining the behaviors of a machine ensemble that operates in an environment as follows.

Definition 4. *Given a machine ensemble* $\mathcal{E} = (J \cup \{e\}, \{M_j\}_{j\in J}, E, src)$ *with environment constraint* c_e, *the corresponding transition system is defined as a pair* $\mathcal{E}_{c_e} = (S^{\mathcal{E}} \times D^{\mathcal{E}}_i, \longrightarrow_{\mathcal{E}_{c_e}})$, *where the transition relation* $\longrightarrow_{\mathcal{E}_{c_e}}$ *is defined by* $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'})$ *iff a machine ensemble in state* $\boldsymbol{s}$ *and with input* $\boldsymbol{i}$ *from the environment has a transition to state* $\boldsymbol{s'}$, *and the environment can generate output* $\boldsymbol{i'}$ *in the next step:*

$$(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'}) \iff \exists \boldsymbol{o}\, ((\boldsymbol{i}, \boldsymbol{s}), (\boldsymbol{s'}, \boldsymbol{o})) \in \delta_{\mathcal{E}} \wedge c_e(\boldsymbol{i'}).$$

$Paths(\mathcal{E}_{c_e})_{(\boldsymbol{s},\boldsymbol{i})}$ *denotes the set of all infinite sequences* $(\boldsymbol{s}, \boldsymbol{i}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s'}, \boldsymbol{i'}) \longrightarrow_{\mathcal{E}_{c_e}} (\boldsymbol{s''}, \boldsymbol{i''}) \longrightarrow_{\mathcal{E}_{c_e}} \cdots$ *of transition steps starting in state* $(\boldsymbol{s}, \boldsymbol{i})$.

Let $\mathcal{E}$ be a machine ensemble with environment constraint c_e, AP a set of *atomic propositions*, and $L : S^{\mathcal{E}} \times D^{\mathcal{E}}_i \to \mathcal{P}(AP)$ a *labeling function* that assigns to each state $(\boldsymbol{s}, \boldsymbol{i}) \in S^{\mathcal{E}} \times D^{\mathcal{E}}_i$ the set $L(\boldsymbol{s}, \boldsymbol{i})$ of atomic propositions that hold in $(\boldsymbol{s}, \boldsymbol{i})$. Then $(\mathcal{E}_{c_e}, L) = (S^{\mathcal{E}} \times D^{\mathcal{E}}_i, \longrightarrow_{\mathcal{E}_{c_e}}, L)$ is the *Kripke structure* associated to $(\mathcal{E}, c_e, L)$.

4 Overview of the PALS Asynchronous Model

This section gives an overview of the asynchronous PALS transformation of a synchronous machine ensemble.

The type of time (discrete or dense) is a parameter of the model. For simplicity and fullest generality, we will assume that all is done in $\mathbb{R}_{\ge 0}$. A basic assumption is that a clock synchronization algorithm is executing "in the background" and guarantees that the difference between the time of a local clock and "real" global

time is always *strictly less* than a given bound ϵ. To reason about clock drift in a general way, we assume a monotonic and continuous[3] *ϵ-drift clock function* $c_j : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ for each machine M_j that assigns to each global instant r the local clock value $c_j(r)$ satisfying $|c_j(x) - x| < \epsilon$.

The shortest, respectively longest, time required for *processing input, executing a transition, and generating output* is assumed to be, respectively, α_{min} and α_{max} with $0 \leq \alpha_{min} \leq \alpha_{max}$. The *message transmission time* is assumed to always be greater than or equal to a minimum value $\mu_{min} \geq 0$, and smaller than or equal to some maximum time value $\mu_{max} \geq \mu_{min}$. The constants $\Gamma = (\epsilon, \alpha_{min}, \alpha_{max}, \mu_{min}, \mu_{max})$ make up the *performance parameters* of the PALS transformation.

Given an ensemble $\mathcal{E}$ and performance parameters Γ, the asynchronous system $\mathcal{A}(\mathcal{E}, \Gamma)$ is made up of a J-indexed family of objects and an environment object, with each object behaving like an "asynchronous typed machine" whose inputs and outputs are received and sent by *asynchronous message passing*. The system is supposed to execute in rounds according to "ticks" of a "logical clock." Let T denote the *period* of the logical clock, and let $t_i = i \cdot T$. Each object j is equipped with two timers: `roundTimer` is a timer that expires at each tick of the logical clock (as perceived by c_j), and `outputBackoffTimer` is used to ensure that output from a machine is not sent into the network too early. The behavior of each object can be summarized as follows:

- When `roundTimer` expires, input from the input buffer is read, a transition is executed, the generated output is put in the output buffer, and the timer is reset to T.
- When the `outputBackoffTimer` timer expires, the messages in the output buffer are sent, *provided that they have been generated.* If the execution of the transition generating the outgoing messages is not yet finished when the timer expires, then the messages are sent as soon as the execution of the transition is finished. This timer is started and set to $dly_{out} = 2 \cdot \epsilon \texttt{ monus } \mu_{min}$, where `monus` is defined by $x \texttt{ monus } y = \max(0, x - y)$, each time the `roundTimer` expires.

The "time-line analysis" for object j in $\mathcal{A}(\mathcal{E}, \Gamma)$ is therefore as follows:

1. At each *local* logical clock tick (that is, when the local clock c_j shows t_i), the object gets the messages from the input buffer, executes a transition, and puts the output messages in the output buffer. This starts somewhere in the global time interval $(t_i - \epsilon, t_i + \epsilon)$ for round i. This process may end at any global time in the interval $(t_i - \epsilon + \alpha_{min}, t_i + \epsilon + \alpha_{max})$.
2. Since the messages cannot be sent into the network before the back-off timer expires, and before the messages are "ready," the messages from the output buffers are therefore sent into the network at a global time t such that $\max(t_i + \epsilon - \mu_{min},\ (t_i - \epsilon) + \alpha_{min}) < t < \max(t_i + 3 \cdot \epsilon - \mu_{min},\ (t_i + \epsilon) + \alpha_{max})$.

[3] PALS is defined for clock functions that are *piecewise* continuous (see [9]), but due to lack of space, we present in this paper the simpler case when local clocks are continuous.

3. At any global time in the time interval $(\max(t_i + \epsilon, (t_i - \epsilon) + \alpha_{min} + \mu_{min}),\ \max(t_i + 3 \cdot \epsilon - \mu_{min}, (t_i + \epsilon) + \alpha_{max}) + \mu_{max})$, a message could arrive at an object, at which time it is entered into the object's input buffer.

We must ensure that messages generated for round $i + 1$ should be received sometime in the global time interval $(t_i + \epsilon, t_i + T - \epsilon)$. This implies that the PALS period T must satisfy $T \geq \mu_{max} + 2 \cdot \epsilon + \max(2 \cdot \epsilon - \mu_{min}, \alpha_{max})$.

Note finally that, although this paper presents the "optimal" PALS transformation, all correctness results hold as long as the backoff timer is always initialized to a value $b \geq 2 \cdot \epsilon$ `monus` μ_{min} and the PALS period $T \geq \mu_{max} + 2 \cdot \epsilon + \max(b, \alpha_{max})$, which both hold in the avionics case study in Section 7.

5 PALS Formal Model in Real-Time Maude

This section presents the formal specification of the asynchronous PALS system $\mathcal{A}(\mathcal{E}, \Gamma)$ as a rewrite theory in Real-Time Maude.

The state of machine j has sort S_j. For convenience, we add a supersort `State` of all such states. It may take some time to compute the next local state of a machine. During this transition computation time, the local state has the value `[`s`,` t`]`, where s is the next state, and t is the time remaining until the execution of the transition is finished. Such a term `[`s`,` t`]` is called a *delayed state* and has the sort `DlyState`:

```
sorts State DlyState .    subsorts S_1 ... S_|J| < State < DlyState .
op [_,_] : State Time ->  DlyState [ctor right id: 0] .
```

Likewise, during the execution of a transition generating the messages for the next round, these messages are not yet ready to be sent, and hence the output buffer has the value `[`$msgs$`,` t`]`, which we call a *delayed configuration*:

```
sort DlyConfiguration .   subsort Configuration < DlyConfiguration .
op [_,_] : Configuration Time ->  DlyConfiguration [ctor] .
```

We also introduce a supersort `Data` of the sorts $D_1 \ldots D_n$ of the data in the wires.

Each machine M_j, and the environment, is translated into an object instance of a subclass $C_{[j]}$ (resp., `Env`) of the class `Machine` declared as follows:

```
class Machine | state : DlyState,  clock : Time,  inBuffer : MsgConfiguration,
                outBuffer : DlyConfiguration,    roundTimer : Time,
                outputBackoffTimer : TimeInf,    localWiring : LocalWiring .
class C_1 .  ...  class C_k .  class Env .  subclass C_1 ... C_k Env < Machine.
```

Several typed machines, say, $M_{j_1}, \ldots, M_{j_r}$, can be of the same type, and can therefore belong to the same subclass, i.e., $C_{[j_1]} = \cdots = C_{[j_r]}$. The `state` attribute denotes the local state of the machine (the `state` attribute for the environment has the value `*`). The `inBuffer` attribute is the buffer of incoming messages. `outBuffer` is the output message buffer. The timers `roundTimer` and

outputBackoffTimer were explained in Section 4. The clock attribute shows the value of the local clock of the object, and the localWiring attribute assigns to each output port number the set of input ports to which this port is connected. However, notice that here a connection is only a *reference* for asynchronous message passing, and not a real "wired" connection as in the synchronous model.

Since in a real application the actual timing of inputs from the environment may be quite unpredictable, the environment outputs must be buffered and synchronized by the object wrapping it in the exact same way as all other machines.

Messages have the form to j from j' (p, d) where $j, j' \in J \cup \{e\}, 1 \leq p \leq n_j$, and $d \in D^j_{i_p}$. Therefore, p is the pth input port of the intended recipient j, where output data d from j' is to be received.

The following actions are modeled by *instantaneous* rewrite rules:

1. Receive an incoming message and put it into the inBuffer.
2. When the roundTimer expires, the inBuffer is emptied, a transition is applied, and the output is put into the outBuffer.
3. When the outputBackoffTimer expires, if the output is ready, then the contents in the output buffer are sent into the network, with message delays.
4. Otherwise, as soon as the output is ready *after* the outputBackoffTimer has expired, the generated output is sent into the network.

In the following rule, corresponding to action (1) above, a message is received by an object and is inserted into its inBuffer:

```
vars j j' : Oid .  var B : MsgConfiguration .  var p : Nat .  var d : Data .
rl [receiveMsg] :  (to j from j' (p, d))  < j : Machine | inBuffer : B >  =>
                   < j : Machine | inBuffer : B (to j from j' (p, d)) > .
```

When the roundTimer expires, the messages B in the inBuffer are read, and a transition is taken (action (2)). Since different classes will have different transitions, executing transitions is modeled by a *family* of rewrite rules, one for each class $C_{[j]}$. The resulting state and messages are delayed by a value $\alpha_{min} \leq$ X-DLY $\leq \alpha_{max}$. In addition, the roundTimer is reset to the round time T, and the outputBackoffTimer is set to $2 \cdot \epsilon$ monus μ_{min}:

```
var X-DLY : Time .     vars S NEXT-STATE : State .     var W : LocalWiring .
var d_{j_1} : D^j_{o_1} .    ...    var d_{j_{m_j}} : D^j_{o_{m_j}} .
crl [applyTrans] :
    < j : C_[j] | inBuffer : B, roundTimer : 0, state : S, localWiring : W >
   =>
    < j : C_[j] | inBuffer : none, state : [NEXT-STATE, X-DLY], roundTimer : T,
                 outputBackoffTimer : (2 · ε monus μ_min),
                 outBuffer : [makeMsg(j, W, (d_{j_1}, ..., d_{j_{m_j}})), X-DLY] >
   if X-DLY >= α_min and X-DLY <= α_max
      /\ ((vect_[j](B), S), (NEXT-STATE, (d_{j_1}, ..., d_{j_{m_j}}))) ∈ δ_{M_j} .
```

The function $vect_{[j]}$(B) maps B to the appropriate vector of inputs $(d_1, \ldots, d_{n_j})$. makeMsg is the function that looks at the local wiring diagram W, takes the vector

of output data from j, and produces the set of messages for the machines and environment getting inputs from that wire.

Due to lack of space, we refer to [9] for the rewrite rules defining the sending of outputs from the output buffer into the network (actions (3) and (4) above).

Since the environment class `Env` is a subclass of `Machine`, the environment inherits the rules for receiving and sending messages. The "machine" rules for reading the input buffer and executing a transition are replaced by one rule that consumes the messages in the input buffer, and generates the output nondeterministically, ensuring that the environment constraint c_e is satisfied:

```
var D1 : D^e_o1 .   ...  var DME : D^e_ome .
crl [consumeInputAndGenerateOutput] :
    < e : Env | inBuffer : B, roundTimer : 0, wiring : W >
   =>
    < e : Env | inBuffer : none, roundTimer : T,
                outputBackoffTimer : (2 · ε monus μmin),
                outBuffer : [makeMsg(e,W,(D1, ..., DME)), X-DLY] >
  if ce(D1, ..., DME)  /\  X-DLY >= αmin and X-DLY <= αmax .
```

Time Behavior. The global state of the system has the form $\{C; t\}$, where C is the configuration consisting of the objects and messages in the asynchronous system, and t is the global time. The tick rule, advancing the global time in the system, is the following slight modification of the "usual" tick rule for object-oriented systems [11]:

```
var C : Configuration .  vars T T' : Time .
crl [tick] :
    {C ; T} => {delta(C,T,T') ; T + T'} in time T' if T' <= mte(C,T) .
```

`delta` is the function that defines how the *passage of time affects the state*, and `mte` is the function that defines the *smallest time until a timer becomes zero*. We refer to [9] for an explanation of the definition of these functions.

Initial States. We define the initial states of the system to start at time $T_0 = T - \epsilon$. At global times $(i \cdot T) + T_0$, for all $i \in \mathbb{N}$, the state components are undelayed and consistent, and all the input buffers are full. Therefore, in the initial state, for each object j, the `clock` attribute is $c_j(T - \epsilon)$, the `outputBackoffTimer` is turned off, the `roundTimer` is initialized to $T - c_j(T_0)$, the `state` attribute is an initial state of the expected sort, the `inBuffer` is full of messages, the `outBuffer` is empty, there are no messages in transit in the network, and the initial input $(d^e_{o_1}, \ldots, d^e_{o_{m_e}})$ from the environment satisfies the environment constraint c_e.

6 Correctness and Optimality of PALS

This section establishes a precise relationship between the synchronous composition $\mathcal{M}_\mathcal{E}$ of an ensemble $\mathcal{E}$ and the asynchronous model $\mathcal{A}(\mathcal{E}_{c_e}, \Gamma)$.

Definition 5. *A state* $\{C\,;t\}$ *in* $\mathcal{A}(\mathcal{E})$ *is called* stable *iff all input buffers in* C *are full, all output buffers are empty, and there is no message "in transit" in* C.

A stable state corresponds to a state $(\boldsymbol{s}, \boldsymbol{i})$ in $M_{\mathcal{E}}$ in the expected way (see [9] for a more detailed definition):

Definition 6. *The function* $sync : Stable(\mathcal{A}(\mathcal{E})) \rightarrow S^{\mathcal{E}} \times D_i^{\mathcal{E}}$ *maps each stable state of the asynchronous model to a state of the synchronous system as follows:*

- *The local state of each object* j*, given in the object's* `state` *attribute, determines the local state in* M_j*. This is well defined, since in a* stable *state, the* `state` *attribute is not a "delayed" value.*
- *The messages in the input buffers determine the state of the environment input and feedback wires using the functions* $fout_j$*,* $j \in J$.

The requirement that the messages in the `inBuffers` *in the initial states are wiring consistent ensures that sync is well-defined for all reachable stable states.*

Definition 7. *For* $\{C_i\,;\,t_i\}$ *a state in* $\mathcal{A}(\mathcal{E})$ *reachable from some initial state of the form described in Section 5, a* path *in* $\mathcal{A}(\mathcal{E})$ *is an infinite or nonextensible finite sequence*

$$\{C_i\,;\,t_i\} \longrightarrow \{C_{i+1}\,;\,t_{i+1}\} \longrightarrow \{C_{i+2}\,;\,t_{i+2}\} \longrightarrow \cdots$$

of one-step rewrites in $\mathcal{A}(\mathcal{E})$*. The above path is called* time-diverging *if and only if for each time value* $t \in \mathbb{R}_{\geq 0}$*, there exists a* $q \in \mathbb{N}$ *such that* $t_q \geq t$*. We denote by* $Paths(\mathcal{A}(\mathcal{E}))_{\{C;t\}}$ *the set of paths in* $\mathcal{A}(\mathcal{E})$ *starting in* $\{C\,;\,t\}$*, and denote by* $TDPaths(\mathcal{A}(\mathcal{E}))_{\{C;t\}}$ *the set of time-diverging paths.*

Theorem 1 in [9] proves that any finite rewrite sequence from an initial state described in Section 5 can be extended into a time-diverging path.

The following theorem states the semantic equivalence between satisfaction of temporal logic properties in $\mathcal{E}_{c_e}$ and in $\mathcal{A}(\mathcal{E})$. Of course, $\mathcal{A}(\mathcal{E})$ has many "unstable" states that do not correspond to any states in $\mathcal{E}_{c_e}$. Therefore, a temporal logic property $\varphi \in CTL^* \setminus \{\bigcirc\}\,(AP)$ of $\mathcal{E}_{c_e}$, when evaluated in $\mathcal{A}(\mathcal{E})$ has somehow to be restricted to the stable states in order to be meaningful. This is accomplished by a related formula φ_{stable} as explained below. The Kripke structure associated to $\mathcal{A}(\mathcal{E})$ is denoted $(\mathcal{A}(\mathcal{E}), L') = (\mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}}, \longrightarrow^1_{\mathcal{A}(\mathcal{E})}, L')$, where $\mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}}$ is the set of E-equivalence classes of ground terms of sort `GlobalSystem` for E the equations of the theory $\mathcal{A}(\mathcal{E})$, $\longrightarrow^1_{\mathcal{A}(\mathcal{E})}$ is the one-step rewrite relation between such equivalence classes, and L' is a labeling function satisfying the requirements explained in the theorem below.

Theorem 1. *Given a formula* $\varphi \in CTL^* \setminus \{\bigcirc\}(AP)$*, and assuming that a new state predicate* $stable \notin AP$ *characterizing stable states has been defined, then there is a formula* $\varphi_{stable} \in CTL^* \setminus \{\bigcirc\}(AP \cup \{stable\})$ *defined as follows:*

$$\begin{aligned}
a_{stable} &= a, \text{ for } a \in AP \\
(\neg\,\varphi)_{stable} &= \neg\,(\varphi_{stable}) \\
(\varphi_1 \wedge \varphi_2)_{stable} &= {\varphi_1}_{stable} \wedge {\varphi_2}_{stable} \\
(\varphi_1\; U\; \varphi_2)_{stable} &= (stable \rightarrow {\varphi_1}_{stable})\; U\; (stable \wedge {\varphi_2}_{stable}) \\
(\forall\,\varphi)_{stable} &= \forall\,\varphi_{stable}
\end{aligned}$$

such that for each stable state s in $\mathcal{A}(\mathcal{E})$ *reachable from initial states defined in Section 5, we have*

$$(\mathcal{A}(\mathcal{E}), L'), s \models \varphi_{stable} \iff (\mathcal{E}_{c_e}, L), sync(s) \models \varphi,$$

where $CTL^{*}\backslash\{\bigcirc\}(AP \cup \{stable\})$ *formulas are interpreted in* $(\mathcal{A}(\mathcal{E}), L')$ *under the* time-diverging path semantics, *and where* $L' : \mathcal{T}_{\mathcal{A}(\mathcal{E})_{\texttt{GlobalSystem}}} \to \mathcal{P}(AP \cup \{stable\})$ *satisfies* $L'(s) = L(sync(s)) \cup \{stable\}$ *when s is a stable state, and* $stable \notin L'(s)$ *otherwise.*

Finally, we show that $T = 2\epsilon + \mu_{max} + \max(\alpha_{max}, 2\epsilon - \mu_{min})$ is the smallest possible period for PALS, in the sense made precise below. Proofs are given in [9].

Proposition 1. *Assume that each object reads its input buffer at its* local *time* t_0*, and at that time starts performing a transition and generating new messages. To ensure that all such generated messages are read by all other objects at or before their* local *times* $t_0 + T$*, we must have* $T \geq 2\epsilon + \mu_{max} + \alpha_{max}$.

Proposition 1 proves optimality of T when $\alpha_{max} \geq 2\epsilon - \mu_{min}$. For the converse, and highly unlikely, case where $2\epsilon - \mu_{min} > \alpha_{max}$, it is harder to claim a "general" optimality result. PALS uses a backoff timer to avoid that messages arrive too early. One could imagine variations of PALS where no such backoff timers are used, but where messages are instead equipped with, e.g., sequence numbers denoting the round in which they were generated. However, if we want to ensure that each message arrives in the right round by using backoff timers, then the backoff timers must be set to at least $2\epsilon - \mu_{min}$:

Proposition 2. *To ensure that a message generated in round i (i.e., at* local *time* $i \cdot T$*) does not arrive too early (i.e., is read by another object at that object's local time* $i \cdot T$*), the message must not be sent before* local *time* $i \cdot T + 2\epsilon - \mu_{min}$.

The optimality of the period follows immediately:

Proposition 3. *If each message for round* $i+1$ *is sent no earlier than at* local *time* $i \cdot T + 2\epsilon - \mu_{min}$*, then we must have* $T \geq 4\epsilon + \mu_{max} - \mu_{min}$ *to ensure that each object has received these messages at its* local *time* $i \cdot T + T$.

7 An Avionics Case Study

To investigate the benefits of PALS for model checking, we have defined in Maude and Real-Time Maude, respectively, a synchronous version and a simplified asynchronous PALS version of an avionics system. The *active standby* system is in essence a synchronous design, but must be realized as a distributed system for fault tolerance reasons. Our models are based on an AADL model developed by Abdullah Al-Nayeem at UIUC of a similar specification developed by Steve Miller and Darren Cofer at Rockwell-Collins (see [10]). The executable specifications are available at `http://www.ifi.uio.no/RealTimeMaude/PALS` and are described in detail in [9].

The Active Standby System. In *integrated modular avionics* (IMA), a cabinet is a chassis with a power supply, internal bus, and general purpose computing, I/O, and memory cards. Aircraft applications are implemented using the resources in the cabinets. There are always two or more cabinets that are physically separated on the aircraft so that physical damage does not take out the computer system. The *active standby* system considers the case of two cabinets and focuses on the logic for deciding which side is *active*. The two sides receive inputs through communication channels. Each side could fail, and a failed side can recover after failure. In case one side fails, the non-failed side should be the active side. In addition, the pilot can toggle the active status of these sides. The full functionality of each side is dependent on these two sides' perception of the availability of other system components. Only a fully functional/available side should be active. The paper [10] states five important properties that the system must satisfy.

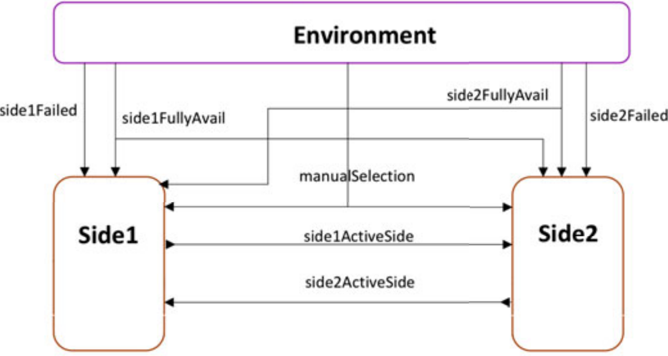


Fig. 2. The architecture of the active standby system

The architecture of the system is shown in Figure 2. The design of the active standby system is globally synchronous: each time Environment dispatches, it nondeterministically sends 5 Boolean values, one through each of its ports, with the following constraints: (i) two sides cannot fail at the same time, and (ii) a failed side cannot be fully available. Therefore, in each round, the environment can send any one of 16 different 5-tuples. The connections between the two sides are "delayed" connections; a message sent in one round is read by the other side in the next round.

The Formal Models. We have defined in Maude the synchronous composition of the three components in the active standby system, and have defined in Real-Time Maude a much simplified PALS asynchronous model of the active standby system. The simplifications in the PALS model include: (i) perfect and perfectly synchronized clocks, (ii) discrete time, (iii) the execution time of a transition, including reading from the input buffer and writing to the output buffer, and

the minimum network delay are both 0, and (iv) the maximum network delay is a parameter `maxMsgDelay` of the system. The behavior of the asynchronous system is then:

– The length of each PALS period is `maxMsgDelay` + 2.
– When a new round starts, an object reads the incoming messages from its input buffer, performs the transition, thereby changing its internal state and generating output messages, which are placed in the object's output buffer.
– One time unit after the start of the PALS period, each object sends its output messages from the output buffer into the network in *one* step.
– When an object receives a message, the message is stored in its input buffer.

Model Checking the Synchronous and Asynchronous Models. We have compared the number of reachable states in our two models, as well as the execution times for model checking analysis. We have chosen an invariant to compare the model checking performance in the synchronous and asynchronous models, since model checking an invariant requires exhaustive search of all reachable states and is therefore not subject to peculiarities in the search strategy.

In the *synchronous model*, the number of states reachable from the initial state is *185*. Both reachability analysis and LTL model checking of the five important properties take less than a second on a 1.86 GHz server with 8GB RAM.

With *no messaging delay*, we could model check the *asynchronous model* in 33 minutes. However, with *maximal messaging delay* 1, exhaustive state space exploration of the asynchronous model was *aborted* by the operating system after two hours, due to the execution using up too much memory. We have also experimented with restricting the environment to 8, respectively 12, possible outputs. The number of reachable states and the execution times for the search command that searches for a state violating an invariant are given in the following table:

Model	Max.msg.dly	8 env. possibilities		12 env. poss.		16 env. poss.	
		# states	ex.time	# states	ex.time	# states	ex.time
Synchr.	n/a	47	0.04 sec.	107	0.1 sec.	185	0.2 sec.
Asynchr.	0	243,360	30 sec.	1,041,376	190 sec.	3,047,832	2000 sec.
Asynchr.	1	349,856	52 sec.	1,496,032	420 sec.	aborted	

The system is not particularly large: 10 messages are sent in each round; the number of internal states of each machine is bounded by 36; the data in each message is either a Boolean value or a number between 0 and 2; time is discrete; executions are instantaneous; message delays are either 0 or 1; and there are no clock skews. The main factor contributing to the state space explosion is the great number of interleavings caused by the intrinsic concurrency of the asynchronous system, since there are 10! different orders in which messages in one round can be received.

8 Related Work

We first explain how this work is related to other work on PALS involving our colleagues at Rockwell-Collins and at UIUC [10,1,14]. The PALS transformation

itself and its optimal period, as well as the active standby example, are also presented in [10,1,14]. The main new contributions of the work presented here are the formal specification of PALS as a parametrized real-time rewrite theory, the proof of correctness of such a formal model, the mathematical justification of the method by which the verification of temporal logic properties of the DRTS thus obtained can be reduced to the verification of such properties on the simpler synchronous model, and the formal analyses of the Maude models of the two versions of the active standby system.

Models of distributed computation are classified into: (i) *synchronous* models, which operate in a lock-step fashion; and (ii) *asynchronous* ones, where there is no a priori bound on message delays. Our ensemble notion is an automata-theoretic synchronous model quite similar to other models, e.g., the synchronous systems of [15], and the synchronous Mealy machine model of [17]. The PALS pattern can be seen as part of a broader body of work on so-called *synchronizers*, which allow synchronous systems to be *simulated* by asynchronous ones. Very general synchronizers such as those in [2] place no a priori bounds on message delays, so that *physical time* in the original synchronous system is simulated by *logical time*[4] in its asynchronous counterpart. More recent work has developed synchronizers for the Asynchronous Bounded Delay (ABD) Network model [3,16], in which a bound can be given for the delay of any message transmission from any process to any other process. PALS can be understood as a synchronizer that also assumes the ABD model (plus clock synchronization) but provides *hard real-time guarantees* needed for embedded systems. The main differences between the synchronizers in [3,16] and PALS are:

1. PALS assumes that a clock synchronization algorithm with a skew bound ϵ is running in the underlying infrastructure. Instead, for the synchronizers in [3,16,15] a clock synchronization algorithm is part of the synchronizer.
2. In the synchronizers in [3,16], two nodes could be in completely different (local) rounds at the same global physical time, and there may be no global physical time at which all nodes are in the same round. This lack of "physical time synchronization" is unsatisfactory for distributed embedded systems. In contrast, in PALS, at any moment in (global) physical time, each node is either in round i or in round $i+1$, and in each round there are "stable" states in which all components are in the same round.
3. There is also a period optimality result in [16] which has a counterpart in the PALS's optimal period. However, optimality in PALS ensures synchrony in physical time while this cannot be ensured by the synchronizer in [16].

Other works relate synchronous and asynchronous models in various ways. Work by Tripakis et al. [17] relates a synchronous Mealy machine model to a loosely timed triggered architecture with local clocks that can advance at different rates with no clock synchronization. The main difference with PALS is that it does not seem possible to give hard real time bounds for the behavior of the asynchronous system realization. The Globally Synchronous Locally

[4] That is, an assignment of logical clocks to processes in the style of [7], whose values need not reflect physical time.

Asynchronous (GALS) Architecture, e.g., [5,6,12] is aimed at a broader class of systems than those modeled by PALS: GALS systems may be widely distributed and it may not be possible to assume that all message communication delays are bounded, although such delays may be bounded within a synchronous subdomain. The main difference between GALS and PALS is that no hard real-time guarantees can be given for a GALS implementation.

The ABD Network model used by the synchronizers in [3,16] and by PALS places stringent demands on an actual network design to guarantee bounded time delivery of messages and bounded clock skew. These demands have stimulated research on network architectures. Rushby [13] gives a detailed discussion of several of these architectures.

9 Conclusions

This work has presented a formal specification of the PALS architectural pattern for obtaining correct-by-construction distributed real-time systems from their synchronous designs under given performance assumptions on the underlying infrastructure. Using the PALS formal model we have given proofs of correctness of PALS, and of optimality of the PALS period; and we have based on such proofs a method to verify temporal logic properties of the DRTS so obtained by verifying such properties on its much simpler synchronous design. We have also illustrated this method's usefulness by means of an avionics case study. We believe that PALS, as a formalized architectural pattern that greatly reduces system complexity, can substantially increase system quality and can greatly reduce the cost of design, verification, and implementation of distributed real-time systems; and also the cost of certifying highly critical systems of this kind.

Acknowledgments. This work is part of a broader collaboration with Steve Miller and Darren Cofer at Rockwell-Collins and with Lui Sha, Abdullah Al-Nayeem, and Mu Sun at UIUC on the PALS architecture. The PALS ideas have been developed in close interaction with all these people, who have provided very useful comments on earlier versions of this work. We also thank the anonymous reviewers for helpful comments on an earlier version of this paper. We gratefully acknowledge funding for this research from the Rockwell-Collins corporation. Partial support has also been provided by the National Science Foundation under Grants IIS 07-20482 and CNS 08-34709, by the Boeing Company under Grant C8088-557395, and by the Research Council of Norway.

References

1. Al-Nayeem, A., Sun, M., Qiu, X., Sha, L., Miller, S.P., Cofer, D.D.: A formal architecture pattern for real-time distributed systems. In: Proc. RTSS 2009. IEEE, Los Alamitos (2009)
2. Awerbuch, B.: Complexity of network synchronization. J. ACM 32(4), 804–823 (1985)

3. Chou, C.-T., Cidon, I., Gopal, I.S., Zaks, S.: Synchronizing asynchronous bounded delay networks. IEEE Trans. Commun. 38(2), 144–147 (1990)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Garavel, H., Thivolle, D.: Verification of GALS systems by combining synchronous languages and process calculi. In: Păsăreanu, C.S. (ed.) SPIN Workshop. LNCS, vol. 5578, pp. 241–260. Springer, Heidelberg (2009)
6. Girault, A., Ménier, C.: Automatic production of globally asynchronous locally synchronous systems. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491. Springer, Heidelberg (2002)
7. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM 21(7), 558–565 (1978)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
9. Meseguer, J., Ölveczky, P.C.: Formalization and correctness of the PALS architectural pattern for distributed real-time systems. Technical Report at CS dept., University of Illinois at Urbana-Champaign (2010), http://hdl.handle.net/2142/17089
10. Miller, S.P., Cofer, D., Sha, L., Meseguer, J., Al-Nayeem, A.: Implementing logical synchrony in integrated modular avionics. In: Proc. 28th Digital Avionics Systems Conference. IEEE, Los Alamitos (2009)
11. Ölveczky, P.C., Meseguer, J.: Semantics and pragmatics of Real-Time Maude. Higher-Order and Symbolic Computation 20(1-2), 161–196 (2007)
12. Potop-Butucaru, D., Caillaud, B.: Correct-by-construction asynchronous implementation of modular synchronous specifications. Fundam. Inform. 78(1), 131–159 (2007)
13. Rushby, J.M.: Bus architectures for safety-critical embedded systems. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 306–323. Springer, Heidelberg (2001)
14. Sha, L., Al-Nayeem, A., Sun, M., Meseguer, J., Ölveczky, P.C.: PALS: Physically asynchronous logically synchronous systems. Technical report, University of Illinois at Urbana-Champaign (2009), http://hdl.handle.net/2142/11897
15. Tel, G.: Introduction to Distributed Algorithms. U.P., Cambridge (1994)
16. Tel, G., Korach, E., Zaks, S.: Synchronizing ABD networks. IEEE Trans. Networking 2(1), 66–69 (1994)
17. Tripakis, S., Pinello, C., Benveniste, A., Sangiovanni-Vincentelli, A., Caspi, P., DiNatale, M.: Implementing synchronous models on loosely time triggered architectures. IEEE Trans. on Computers 1 (2008)

Automated Multiparameterised Verification by Cut-Offs

Antti Siirtola

University of Oulu, Department of Information Processing Science,
P.O. Box 3000, 90014 University of Oulu, Finland
antti.siirtola@oulu.fi

Abstract. We consider multiparameterised process algebraic verification, where parameters are sets and binary relations over these sets used to respectively denote the sets of the identities of replicated components and the topology of a system. There is a cut-off result that enables such a parameterised verification task to be reduced to a finite set of finite-state ones, but no practical way to perform reduction, i.e. to compute the parameter values up to the cut-offs. The first contribution of the paper is an improved formalism that enables parameterised systems and specifications to be expressed with fewer parameters than before. The second one is a search-tree-based algorithm for computing the parameter values up to the cut-offs. The algorithm detects and discards isomorphic parameter values and is equipped with a heuristic to prune a search tree. The algorithm is implemented and the relevance of the contributions is justified by practical computations.

Keywords: formal verification, parameterised verification, refinement, process algebra, cut-off.

1 Introduction

Parameterised verification is important in practice, as probably all real-life systems can naturally be modelled as (multiply) parameterised finite-state machines, but also theoretically challenging, because the parameterised verification problem (PVP) is undecidable in general [1].

We consider multiparameterised verification, where parameters are type and relation variables used to respectively denote the sets of the identities of replicated components and relationships between components, i.e. the topology of a system. A specification and a system are given as parameterised labelled transition systems, which are constructed using parallel composition and hiding, parameter values are obtained as the model class of a first order formula, where terms are restricted to simple variables and predicates to relation variables, and correctness is understood as the traces refinement. There is a cut-off result that enables such a parameterised verification task to be reduced to a finite set of finite-state ones without changing the answer to the problem, provided that the specification does not involve hiding and the parameter values are specified using

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 321–337, 2010.
© Springer-Verlag Berlin Heidelberg 2010

the universal fragment of first order logic [2]. After the reduction, the verification task can be solved using existing refinement checking tools.

There is also a simple algorithm which does the reduction basically automatically, but its role is just to show the decidability of the problem [2]. The problem is that the algorithm is based on the brute force exploration of the search space, the size of which is exponential in the number of relation variables. Moreover, the formalism suggested in [2] often necessitates the use of relation variables carrying redundant information, because the relations represented by two variables cannot be combined without introducing a new variable.

The first contribution of the paper is an improved formalism that enables parameterised systems and specifications to be expressed with fewer relation variables. The second one is a search-tree-based algorithm for computing the parameter values up to the cut-offs. The algorithm discards isomorphic parameter values and is equipped with a heuristic to prune a search tree.

Isomorphism rejection allows us to reduce the number of remaining computationally expensive refinement checking tasks, so it usually pays off. It is done by converting parameter values to vertex coloured graphs first and then applying the existing algorithms and tools for graph isomorphism [3]. The pruning heuristic, in turn, can have only a positive influence on the running time of the algorithm but its efficiency depends on the topology of a system and the way a modeller describes it. However, as the conditions behind the application of the heuristic are syntactic, it should not be difficult for a modeller to specify a topology in such a way that (s)he gets the most out of the heuristic.

We have implemented the algorithm in a prototype tool. Using the tool, three example systems are analysed and the results show that the overhead of the isomorphism rejection is minimal, running time increases rapidly in the number of relation variables and computation becomes quickly infeasible when the heuristic is disabled. Hence, both the contributions are of practical relevance.

A distinctive feature of our method is that it is fully automated, allows parameterising a system topology, and guarantees successful termination on every input. Probably the closest related work is done by Clarke et al. [4], who study networks of homogeneous fixed-size processes communicating through token passing. Their result applies to systems of any topology but allows components of an only one kind. Moreover, they provide an upper bound for the size of network graphs only, no method for determining the networks up to the cut-off.

Also the behavioural fixed point method of Valmari and Tienari is worth mentioning [5]. It can be seen as a kind of a cut-off result, where the bound for the number of replicated components is determined by repeatedly adding a component to a (partial) system until its behaviour converges. Although the necessary computations can be done using existing tools, there is no clearly defined class of systems on which the method terminates successfully.

Other approaches that enable parameterised verification by cut-offs have been proposed for systems composed of similar fixed-size processes [6,7], systems of a ring topology [8,9,10,11,12], and request-take-release (RTR) systems, where identical processes compete for an access to a fixed number of shared resources

under a prioritised queue policy [13]. Most of the methods allow only a single parameter, the number of replicated components, so they are easy to implement [13,8,7,10,9,11,12]. The multiparameterised case is considered by Emerson and Kahlon only, but neither does their result allow parameterising anything else but the number of replicated components [6]. Moreover, the results concerning the verification of the safety properties of the RTR systems [13] and the systems with conjunctive guards [6] can be obtained with our method, too [2].

Other kinds of parameterised verification methods are based on abstract interpretation or inductive reasoning, but despite few exceptions they are not fully algorithmic or guaranteed to terminate. Moreover, the exceptions [14,15], based on either counter abstraction [14] or infinite-state verification algorithms [15], do not allow parameterising a system topology.

In the next section, our model of computation, an LTS, is reviewed. After that, the improved formalism and the cut-off result are introduced. Sect. 4 presents the new algorithm and provides empirical evidence on its performance. The paper concludes with a brief discussion on future work.

2 Labelled Transition Systems

Our fundamental model of computation is a (finite) labelled transition system (LTS) [16]. Intuitively, an LTS is a graph the vertices of which are called states, the edges are labelled by actions and they are called transitions, and one of the states is marked as the initial one. To introduce LTSs formally, we assume a countably infinite set $\mathbb{A}$ of *atoms*. Tuples of atoms are called *actions*. The empty tuple $()$, also denoted by τ, is called the *invisible* action and the rest of the actions are *visible*. The set of all the visible actions is referred to by $\mathbb{V}$.

Definition 1 (LTS). *An* LTS *is a four-tuple* $(S, \Sigma, R, \hat{s})$*, where* S *is a finite non-empty set of* states*,* $\Sigma \subseteq \mathbb{V}$ *is a finite set of visible actions, an* alphabet*,* $R \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ *is a set of* transitions *and* $\hat{s} \in S$ *is the* initial state*.*

In the analysis of LTSs, we are usually interested in the sequences of visible actions reachable from the initial state. A finite alternating sequence of states and actions, $s_1\mathbf{a}_1 s_2 \ldots \mathbf{a}_{n-1} s_n$, of an LTS $\mathcal{L}$, is a *path in* $\mathcal{L}$ *from* s_1, if $(s_i, \mathbf{a}_i, s_{i+1})$ is a transition of $\mathcal{L}$ for every $i \in \{1, \ldots, n-1\}$. A finite sequence of visible actions is a *trace (of* $\mathcal{L}$*)*, if there is a path in $\mathcal{L}$ from the initial state such that the sequence can be obtained from the path by removing all the states and the invisible actions. The set of all the traces of $\mathcal{L}$ is denoted by $\mathrm{tr}(\mathcal{L})$.

An LTS $\mathcal{L}_2$ is a *traces refinement* of an LTS $\mathcal{L}_1$, denoted by $\mathcal{L}_1 \succeq_{\mathrm{tr}} \mathcal{L}_2$, if $\mathcal{L}_1$ and $\mathcal{L}_2$ have the same alphabet and $\mathrm{tr}(\mathcal{L}_2) \subseteq \mathrm{tr}(\mathcal{L}_1)$ [17]. The LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$ are *traces equivalent*, denoted by $\mathcal{L}_1 =_{\mathrm{tr}} \mathcal{L}_2$, if both of them are traces refinements of each other. Clearly, $\succeq_{\mathrm{tr}}$ is a preorder (i.e. a reflexive and transitive relation) and $=_{\mathrm{tr}}$ an equivalence relation in the set of LTSs.

In process algebraic verification, both the specification and the system are typically modelled as LTSs $\mathcal{L}_{Spec}$ and $\mathcal{L}_{Sys}$, respectively. If $\mathcal{L}_{Sys}$ is a traces refinement of $\mathcal{L}_{Spec}$, then the system cannot do anything more than the specification does.

This way, it is possible to prove absence of some unwanted behaviour. Therefore, the traces refinement is applicable in proving safety properties.

A system modelled as an LTS is typically composed of smaller LTSs representing its parts. Let $\mathcal{L}_i = (S_i, \Sigma_i, R_i, \hat{s}_i)$ be an LTS for both $i \in \{1, 2\}$. The *parallel composition* of LTSs $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted by $(\mathcal{L}_1 \parallel \mathcal{L}_2)$, is a four-tuple $(S_1 \times S_2, \Sigma_1 \cup \Sigma_2, R_\parallel, (\hat{s}_1, \hat{s}_2))$, where $R_\parallel$ consists of all triplets $((s_1, s_2), \mathbf{a}, (s'_1, s'_2))$ such that $\mathbf{a} \in \Sigma_1 \cap \Sigma_2$ and $(s_i, \mathbf{a}, s'_i) \in R_i$ for both $i \in \{1, 2\}$; or $\mathbf{a} \in (\Sigma_i \cup \{\tau\}) \setminus \Sigma_j$, $(s_i, \mathbf{a}, s'_i) \in R_i$ and $s_j = s'_j$, where i, j are different elements in $\{1, 2\}$. This definition results in Hoare type parallel composition [17]. According to it, the LTSs can execute a common visible action $\mathbf{a}$ in the parallel composition if and only if both of them agree on its execution, whereas the invisible actions and the visible actions that are only in the alphabet of another LTS are executed individually.

Before a system LTS is compared against a specification one, the actions irrelevant to the specification are hidden. Let $\mathcal{L} = (S, \Sigma, R, \hat{s})$ be an LTS and Λ a set of visible actions. The LTS $\mathcal{L}$ *after hiding* Λ, is a four-tuple $(S, \Sigma \setminus \Lambda, R_\Lambda, \hat{s})$, denoted by $(\mathcal{L} \setminus \Lambda)$, where R_Λ consists of all tuples $(s, \mathbf{a}, s')$ such that $\mathbf{a} \notin \Lambda$ and $(s, \mathbf{a}, s') \in R$; or $\mathbf{a} = \tau$ and there is $\mathbf{b} \in \Lambda$ such that $(s, \mathbf{b}, s') \in R$. Hence, $(\mathcal{L} \setminus \Lambda)$ is obtained from $\mathcal{L}$ by changing the actions in Λ to the invisible one.

It is easy to see that the structures obtained from LTSs by parallel composition and hiding are LTSs. Moreover, the parallel composition is commutative and associative, which means that we can generalise the parallel composition to any finite non-empty set of LTSs, provided we are interested in the alphabet and traces only. Let $I = \{i_1, i_2, \ldots, i_n\}$ be a finite non-empty index set and $\mathcal{L}_i = (S_i, \Sigma_i, R_i, \hat{s}_i)$ an LTS for every $i \in I$. The *parallel composition* of LTSs in the set $\{\mathcal{L}_i\}_{i \in I}$, denoted by $(\parallel_{i \in I} \mathcal{L}_i)$, can be now defined as the LTS $(\mathcal{L}_{i_1} \parallel \mathcal{L}_{i_2} \parallel \ldots \parallel \mathcal{L}_{i_n})$.

3 Multiparameterised Verification by Cut-Offs

The problem with LTSs is that they cannot naturally express parameterised systems and specifications. As an example, you may consider a shared resource system (SRS) with an arbitrary number of users competing for an access to an arbitrary number of shared resources arranged in the form of a forest [18]. A user gets a read (write) access to a subtree of resources after successfully requesting a read (write) lock on the root of the subtree and a weaker read (write) intention lock, whose purpose is to indicate the existence of the read (write) lock deeper in the tree, on all its ancestors. A resource itself has no mechanism for concurrency control and several users can hold a lock on a resource simultaneously only if all of them have either intention locks or read related locks. Our goal is to formally model the system and the specification and prove that in our construction it is not possible for a user to access a resource if somebody else is writing to it.

3.1 Variables and Valuations

Clearly, SRS cannot be expressed as a single LTS but a different LTS is needed for each parameter value. To overcome the problem, we allow the use of three

kinds of parameters, type, atom and relation variables, in LTSs. The sets of all the atom, type and relation variables are denoted respectively by $\mathbb{X}$, $\mathbb{T}$ and $\mathbb{G}$, and they are assumed to be countably infinite and disjoint. We also allow the use of a constant relation symbol $\doteq$ which represents the normal equality (the minimal reflexive relation) in the set of atoms.

Type variables represent the disjoint, finite, non-empty sets of atoms and they are used to denote the sets of the identities of replicated components of the same kind. We assume that for each type variable T there are infinitely many atoms, denoted by $a_{T,1}, a_{T,2}, \ldots$ and called T*-atoms*, used in the values of T. As the T-atoms represent the identities of the replicated components of the type T, it is natural to assume that the sets of T_1-atoms and T_2-atoms are disjoint whenever T_1 and T_2 are different type variables. We may also assume that there are infinitely many atoms that are not U-atoms for any type variable U.

Atom variables are used to refer to the identities of replicated components. Hence, each atom variable x represents a T-atom for some type variable T, which is specified by a function $\delta : \mathbb{X} \mapsto \mathbb{T}$ (i.e. $\delta(x) = T$). *Relation variables* are used to describe the topology of a system and relationships between the components. Formally, a relation variable Π represents a relation over the set of T_1-atoms and T_2-atoms for some type variables T_1, T_2, which are specified by respectively functions $\delta_1, \delta_2 : \mathbb{G} \mapsto \mathbb{T}$ (i.e. $\delta_1(\Pi) = T_1$ and $\delta_2(\Pi) = T_2$). We assume that for all type variables U_1, U_2 there are infinitely many atom variables y and infinitely many relation variables Ξ such that $\delta(y) = U_1$, $\delta_1(\Xi) = U_1$ and $\delta_2(\Xi) = U_2$.

The values of variables are formally represented as a valuation.

Definition 2 (Valuation). *A* valuation *is a function ϕ from a finite set of atom, type and relation variables such that*

- *for all type variables $T \in \mathrm{dom}(\phi)$, $\phi(T)$ is a finite non-empty set of T-atoms,*
- *for all relation variables $\Pi \in \mathrm{dom}(\phi)$, the type variables $\delta_1(\Pi), \delta_2(\Pi)$ are in $\mathrm{dom}(\phi)$ and $\phi(\Pi) \subseteq \phi(\delta_1(\Pi)) \times \phi(\delta_2(\Pi))$, and*
- *for all atom variables $x \in \mathrm{dom}(\phi)$, the type variable $\delta(x)$ is in $\mathrm{dom}(\phi)$ and $\phi(x) \in \phi(\delta(x))$.*

When modelling in our formalism, you have to identify different classes of components and their relationships (system topology) first, and then represent them with the aid of type and relation variables. In the case of SRS, there are two kinds of components, resources and users, so we pick type variables R and U to represent the sets of their identities, respectively. The relationship between resources is represented by a relation variable $<_R$ such that (r_1, r_2) are related by $<_R$ if and only if r_1 is a proper ancestor of r_2. (Here, an ancestor of a resource can also be the resource itself, and a proper ancestor refers to an ancestor other than the resource itself.) Hence, $<_R$ denotes a proper ancestor relation, which is the transitive closure of a forest.

In other words, we are interested in valuations ϕ with the domain $R, U, <_R$ such that $\phi(R)$ and $\phi(U)$ are finite non-empty sets of respectively R-atoms and U-atoms, and $\phi(<_R)$ is a transitive, irreflexive and asymmetric relation over $\phi(R)$ such that whenever $(r_1, r_3), (r_2, r_3) \in \phi(<_R)$ and $r_1 \neq r_2$ then $(r_1, r_2) \in \phi(<_R)$

or $(r_2, r_1) \in \phi(<_R)$ (i.e. whenever r_1 and r_2 are different proper ancestors of r_3, then r_1 is a proper ancestor of r_2 or vice versa). We write Φ_{Srs} for the set of all such valuations.

3.2 Valuation Formulae

Valuations can be considered the interpretations of the formulae of typed (many-sorted) first order logic where the terms are restricted to atom variables and predicates to relation variables. Therefore, some (infinite) sets of valuations can be obtained as the model classes of such first order formulae. Many (or probably all) practically important system topologies (e.g. trees, rings, fully connected ones) can be represented this way. The first order formulae of the above kind are called valuation formulae (VFs) and defined formally as follows.

Definition 3 (Valuation formula (VF))

1. $\top$ *is an (always-true) VF, and if* x, y *are atom variables and* Π *is a relation variable or the relation symbol* $\doteq$, *then* $x\Pi y$ *is an (elementary) VF.*
2. *If* $\mathcal{C}_1$ *and* $\mathcal{C}_2$ *are VFs and* x *is an atom variable, then* $(\neg\mathcal{C}_1)$ *is a (negated) VF,* $(\mathcal{C}_1 \wedge \mathcal{C}_2)$ *a (conjunctive) VF, and* $(\forall x : \mathcal{C})$ *an ((x-)replicated) VF.*
3. *Only the expressions obtained by finite application of the steps 1,2 are* VFs.

We use standard abbreviations $x \not\Pi\, y$, $(\mathcal{C}_1 \vee \mathcal{C}_2)$, $(\mathcal{C}_1 \rightarrow \mathcal{C}_2)$ and $(\forall x_1, \ldots, x_n : \mathcal{C})$ for VFs $\neg(x\Pi y)$, $(\neg((\neg\mathcal{C}_1) \wedge (\neg\mathcal{C}_2)))$, $((\neg\mathcal{C}_1) \vee \mathcal{C}_2)$ and $(\forall x_1 : \cdots : \forall x_n : \mathcal{C})$, respectively. The *conjunction* of VFs $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n$ is naturally defined as the VF $((\ldots((\mathcal{C}_1 \wedge \mathcal{C}_2) \wedge \mathcal{C}_3) \ldots) \wedge \mathcal{C}_n)$, when $n \geq 1$, and as $\top$, in the case $n = 0$.

A VF $\mathcal{C}'$ is a *sub-VF* (of a VF $\mathcal{C}$), if $\mathcal{C}'$ is a substructure of $\mathcal{C}$. Moreover, a VF $\mathcal{C}$ is called *universal*, if existential quantification is not used, i.e. every replicated sub-VF occurs within an even number of negated sub-VFs in $\mathcal{C}$; *negation-normal*, if negations are only applied to elementary and always-true VFs, i.e. whenever $\neg\mathcal{C}'$ is a sub-VF of $\mathcal{C}$, then $\mathcal{C}'$ is an elementary or an always-true VF; *relation-negated*, if each elementary VF, which involves a relation variable, is negated, i.e. every elementary sub-VF $x\Pi y$, where Π is a relation variable, occurs within an odd number of negated sub-VFs in $\mathcal{C}$; and *unquantified*, if quantification is not used, i.e. $\mathcal{C}$ has no replicated sub-VF.

An atom variable x is *free (in* $\mathcal{C}$*)*, if there is an occurrence of x in $\mathcal{C}$ that is not within an x-replicated substructure, otherwise x is *bound (in* $\mathcal{C}$*)*. A VF is *closed*, if every atom variable is bound in it. The *parameters (of* $\mathcal{C}$*)* are the relation and free atom variables in it and the type variables which $\delta, \delta_1, \delta_2$ assign to the relation and atom variables in $\mathcal{C}$. If ϕ is defined for all the parameters of $\mathcal{C}$, then $[\![\mathcal{C}]\!]_\phi$ denotes a formula obtained from $\mathcal{C}$ by substituting the relation and free atom variables in $\mathcal{C}$ according to ϕ and the normal equality relation $=$ for the relation symbol $\doteq$, and by letting the value of each atom variable x used to index a replicated substructure to range over $\phi(\delta(x))$. The formula $[\![\mathcal{C}]\!]_\phi$ is called the *instance (of* $\mathcal{C}$*) (generated by* ϕ*)* and it is evaluated in the usual way.

We say that a valuation ϕ *satisfies* (or in logical terms *is a model of*) a VF $\mathcal{C}$, if ϕ is defined for all the parameters of $\mathcal{C}$ and $[\![\mathcal{C}]\!]_\phi$ is true. (Infinite) sets of

parameter values can be now expressed as the sets of valuations that satisfy a certain VF. For example, the parameter values of SRS is the set of all valuations ϕ with the domain $\{R, U, <_R\}$ such that ϕ satisfies a VF

$$\begin{aligned}&(\forall r_1 : r_1 \not<_R r_1) \wedge (\forall r_1, r_2 : (r_1 \not<_R r_2 \vee r_2 \not<_R r_1))\wedge\\&(\forall r_1, r_2, r_3 : ((r_1 <_R r_2 \wedge r_2 <_R r_3) \rightarrow r_1 <_R r_3))\wedge\\&(\forall r_1, r_2, r_3 : ((r_1 \neq r_2 \wedge r_1 <_R r_3 \wedge r_2 <_R r_3) \rightarrow (r_1 <_R r_2 \vee r_2 <_R r_1)))\ ,\end{aligned}$$

denoted by $\mathcal{C}_{Srs}$, where r_1, r_2, r_3 are atom variables mapped to R by δ. Hence, Φ_{Srs} is the class of the models of $\mathcal{C}_{Srs}$ that are defined for precisely $R, U, <_R$.

3.3 LTS Schemata

We express parameterised systems and specifications as structures called LTS schemata (LTSCs). However, before we can formally define them, we need to introduce set schemata (SSCs) used to express the parameterised sets of actions.

Definition 4 (Set schema (SSC))

1. *A finite set of actions, where zero or more atom variables are used in place of atoms, is an (elementary) SSC.*
2. *If $\mathcal{S}$ is an SSC and x an atom variable, then $\bigcup_x \mathcal{S}$ is a* (replicated) SSC*.*
3. *Only the expressions obtained by finite application of the steps 1,2 are SSCs.*

Definition 5 (LTS schema (LTSC))

1. *An LTS, where zero or more atom variables are used in place of atoms, is an* (elementary) LTSC*.*
2. *If $\mathcal{P}_1$ and $\mathcal{P}_2$ are LTSCs, $\mathcal{C}$ is an unquantified VF, $\mathcal{S}$ an SSC and x an atom variable, then $(\mathcal{P}_1 \parallel \mathcal{P}_2)$ is a* (parallel) LTSC*, $(\parallel_x \mathcal{P}_1)$ an* ((x-)replicated) LTSC*, $(\mathcal{C} \,\&\, \mathcal{P}_1)$ a (conditional) LTSC, and $(\mathcal{P}_1 \setminus \mathcal{S})$ a* (hiding) LTSC*.*
3. *Only the expressions obtained by finite application of the steps 1,2 are LTSCs.*

The concepts of a *sub-SSC* and a *sub-LTSC* are defined analogously to sub-VFs, and the concepts of a *free* and *bound* atom variable, a *closed* structure, a *parameter* and an *instance (generated by a valuation)* are extended to LTSCs and SSCs as they are defined for VFs. Formally, if $\mathcal{P}$ is an LTSC or SSC, and ϕ a valuation defined for all the parameters of $\mathcal{P}$, then $[\![\mathcal{P}]\!]_\phi$ is defined as follows.

1. The instance of an elementary LTSC (SSC) $\mathcal{P}$ is a structure obtained from $\mathcal{P}$ by substituting $\phi(x)$ for every occurrence of each atom variable x.
2. $[\![\bigcup_x \mathcal{S}']\!]_\phi = \bigcup_{a \in \phi(\delta(x))}([\![\mathcal{S}']\!]_{\phi[a \leftarrow x]})$, where $\phi[a \leftarrow x]$ denotes a valuation otherwise equal to ϕ but maps x to a.
3. $[\![(\mathcal{P}_1 \parallel \mathcal{P}_2)]\!]_\phi = [\![\mathcal{P}_1]\!]_\phi \parallel [\![\mathcal{P}_2]\!]_\phi$.
4. $[\![(\parallel_x \mathcal{P}')]\!]_\phi = \parallel_{a \in \phi(\delta(x))}([\![\mathcal{P}']\!]_{\phi[a \leftarrow x]})$.
5. $[\![(\mathcal{C} \,\&\, \mathcal{P}')]\!]_\phi = \begin{cases} [\![\mathcal{P}']\!]_\phi & \text{, if } [\![\mathcal{C}]\!]_\phi \text{ is true,} \\ \text{an identity LTS } \mathcal{L}_{id} := (\{()\}, \emptyset, \emptyset, ()) & \text{, otherwise.} \end{cases}$
6. $[\![(\mathcal{P}' \setminus \mathcal{S})]\!]_\phi = [\![\mathcal{P}']\!]_\phi \setminus [\![\mathcal{S}]\!]_\phi$.

It is easy to see that the instance of an LTSC is an LTS.

When modelling with LTSCs, we first capture the behaviour of a system into finitely many elementary LTSCs each of which represents the system from the viewpoint of finitely many components. Atom variables are used to refer to replicated components. Then, each elementary LTSC is enclosed within a conditional LTSC that restricts the relationships of the replicated components to those that are actually possible. Each conditional LTSC, in turn, is enclosed within replicated LTSCs such that every atom variable becomes bound, and after that, the replicated LTSCs are put together using parallel LTSCs. Specifications are modelled similarly and finally, the actions irrelevant to the specification are hidden in the system model. For the restrictions of the modelling technique, see Prop. 7 in [2] and the related discussion.

In the case of SRS, we first capture the behaviour of a user u in an elementary LTSC $User_1$ from the viewpoint of reading and writing to a resource r_2 based on the lock on its ancestor r_1. Here, u, r_1, r_2 are atom variables such that $\delta(u) = U$ and $\delta(r_1) = \delta(r_2) = R$, and because r_1 denotes an ancestor r_2, we enclose $User_1$ within $\mathcal{C}_{r1 \leq r2} := (r_1 \doteq r_2 \vee r_1 <_R r_2)$. As the lock requests of a user u on a resource and its ancestors are interdependent, we need to model u from the viewpoint of locking a resource r_2 and its proper ancestor r_1, too. We capture this behaviour in an elementary LTSC $User_2$. As r_1 now represents a proper ancestor of r_2, $User_2$ is enclosed within $\mathcal{C}_{r1 < r2} := r_1 <_R r_2$. Finally, as also the lock requests of distinct users u_1, u_2 on the same resource r are interdependent, the system has to be modelled from this viewpoint as well. For that purpose, we introduce an elementary LTSC $Lock$. Here, r, u_1, u_2 are atom variables such that $\delta(r) = R$ and $\delta(u_1) = \delta(u_2) = U$, and because u_1 and u_2 denote different users, $Lock$ is put inside $\mathcal{C}_{u1 \neq u2} := u_1 \neq u_2$. The details of elementary LTSCs can be found in [18, pp. 61–64], but in the context of this paper it is sufficient to be aware of the structure of the system and the specification.

To build the model of SRS, denoted by $\mathcal{P}_{Srs}$, we enclose the LTSCs described above within replicated LTSCs such that every atom variable becomes bound and finally put the parts together using the parallel LTSC construct. Hence, $\mathcal{P}_{Srs}$ is an LTSC

$$(\mathop{\|}_{u} \mathop{\|}_{r_1} \mathop{\|}_{r_2} \mathcal{C}_{r1 \leq r2} \,\&\, User_1) \parallel (\mathop{\|}_{u} \mathop{\|}_{r_1} \mathop{\|}_{r_2} \mathcal{C}_{r1 < r2} \,\&\, User_2) \parallel (\mathop{\|}_{r} \mathop{\|}_{u_1} \mathop{\|}_{u_2} \mathcal{C}_{u1 \neq u2} \,\&\, Lock) \; .$$

To formalise the specification, note that every illegal behaviour can be traced back to two different users u_1, u_2 that write to a resource r_3 simultaneously based on the locks they have on respectively some ancestors r_1, r_2. Therefore, we first capture the specification in an elementary LTSC $Prop_2$ from the viewpoint of u_1, u_2 accessing r_3 based on the locks on respectively r_1, r_2. Here, r_3 is an atom variable such that $\delta(r_3) = R$, and because u_1 and u_2 denote different users and r_1, r_2 ancestors of r_3, we enclose $Prop_2$ within a VF $\mathcal{C}_{r1 \leq r3} \wedge \mathcal{C}_{r2 \leq r3} \wedge \mathcal{C}_{u1 \neq u2}$. However, to correctly present the specification in the presence of a single user only, we also introduce an elementary LTSC $Prop_1$, which captures the specification from the viewpoint of u_1 accessing r_3 based on the lock on r_1.

Hence, it suffices to put $Prop_1$ inside the VF $\mathcal{C}_{r1 \leq r3}$, and the formal specification can be now expressed as an LTSC

$$\mathcal{Q}_{Srs} := \mathop{\|}_{r_1} \mathop{\|}_{r_2} \mathop{\|}_{r_3} \mathop{\|}_{u_1} \mathop{\|}_{u_2} ((\mathcal{C}_{r1 \leq r3} \,\&\, Prop_1) \parallel (\mathcal{C}_{r1 \leq r3} \wedge \mathcal{C}_{r2 \leq r3} \wedge \mathcal{C}_{u1 \neq u2} \,\&\, Prop_2)) \,.$$

Finally, as the actions related to locking are irrelevant to $\mathcal{Q}_{Srs}$, we hide them in the system model. The set of all the locking actions is represented by a SSC $\mathcal{S}_{lock}$, so the system gets the form $\mathcal{P}_{Srs} \setminus \mathcal{S}_{lock}$. The correctness of SRS can be now stated as the question whether $[\![\mathcal{Q}_{Srs}]\!]_\phi \succeq_{\mathrm{tr}} [\![\mathcal{P}_{Srs} \setminus \mathcal{S}_{lock}]\!]_\phi$ for all valuations ϕ with the domain $\{R, U, <_R\}$ such that ϕ satisfies $\mathcal{C}_{Srs}$. Note also that SRS cannot be handled with the other cut-off methods because it involves both multiple and topology-related parameters.

The formalism presented above is developed from that presented in [2]. The main novelties are the introduction of the functions $\delta, \delta_1, \delta_2$ that assign domains to atom and relation variables and the introduction of a conditional LTSC, which has made it possible to simplify a replicated LTSC. Earlier, the domains of atom and relation variables were not fixed which expanded system descriptions. Moreover, relation variables were not restricted to binary relations and a replicated LTSC was assigned a tuple of atom variables whose value ranged over the value of a relation variable. Because the interdependencies in the values of atom variables can be now expressed with the aid of VFs, there is no need to do that in replicated LTSCs. That is why it is sufficient to use simple replicated LTSCs, where the value of a single atom variable ranges over its domain.

The introduction of a conditional LTSC has also enabled us to restrict the domains of relation variables to binary relations. That is because we believe that binary relations are sufficient to express all practically relevant system topologies and, with the aid of conditional LTSCs, they can be combined to express relationships between any number of components. For example, in the case of SRS, we have combined the relations represented by $<_R$ and $\doteq$ in the VFs $\mathcal{C}_{r1 \leq r2}$, $\mathcal{C}_{u1 \neq u2}$, $\mathcal{C}_{r1 \leq r3} \wedge \mathcal{C}_{r2 \leq r3} \wedge \mathcal{C}_{u1 \neq u2}$ to create new binary and ternary relations. Actually, the problem with the earlier formalism was that you could not create a new relation from existing ones without introducing a new relation variable. For example, SRS modelled in the old formalism uses six relation variables [18], all definable with the aid of the type variables U and R, the relation variable $<_R$ and the constant relation symbol $\doteq$. Hence, the new formalism is not only simpler but allows more natural modelling, too.

3.4 Parameterised Traces Refinement

In general, we consider the following version of PVP.

Problem 6 (Parameterised Traces Refinement (PTR)).

Instance: A closed specification LTSC $\mathcal{Q}$ which has no hiding sub-LTSC, a closed system LTSC $\mathcal{P}$, and a closed universal VF $\mathcal{C}$.

Question: Is $[\![\mathcal{Q}]\!]_\phi \succeq_{\mathrm{tr}} [\![\mathcal{P}]\!]_\phi$ for all valuations ϕ that satisfy $\mathcal{C}$ and are defined for precisely the parameters of $\mathcal{Q}$ and $\mathcal{P}$?

Focusing our attention on closed structures is not a restriction, because without the loss of generality, free atom variables can be bound by introducing new relation variables [2]. However, it is necessary to restrict our attention to universal VFs and specifications that have no hiding sub-LTSC, because otherwise the problem becomes undecidable [2,18]. Hiding is typically needed in the system side only, but the restriction to universal VFs means that we can study only specification-system families that are closed under the removal of a replicated component [2, Lemma 13]. In other words, we can only verify systems and study properties whose behaviour can be modelled from the viewpoint of any two (or more) components connected to each other. That is why our formalism is not (naturally) applicable to systems with a linear, ring or tree topology. However, with the aid of the behavioural fixed-point method [5] this restriction can sometimes be overcome [19]. Nevertheless, the results in [19] bring only more systems within the reach of our cut-off result but do not help in automating its application.

PTR is decidable because there is a cut-off result that establishes upper bounds for (the sizes of) the values of parameters such that the system is correct with respect to the specification for all the parameter values if and only if it is correct for all the parameter values up to the cut-offs [2]. However, before we can represent the result, we need to introduce some new concepts first.

Let ϕ and ψ be valuations, T a type variable and $\mathcal{R}$ an LTSC. We say that ϕ and ψ are *isomorphic*, denoted by $\phi \simeq \psi$, if they have the same domain and can be obtained from each other by a bijective mapping of atoms [2]. The T-degree of $\mathcal{R}$, denoted by $\deg_T(\mathcal{R})$, is the maximum number of the nested sub-LTSCs $(\|_x \mathcal{R}')$ of $\mathcal{R}$ such that $\delta(x) = T$.

We can now represent the cut-off result. It gives an explicit upper bound for (the size of) the values of each type variable generally as the maximum of the T-degrees of the system and the specification. The cut-offs for the values of type variables implicitly establish upper bounds for the values of relation variables as well and guarantee that only finitely many non-isomorphic valuations (i.e. refinement checks) remain.

Theorem 7. *Let $\mathcal{Q}, \mathcal{P}, \mathcal{C}$ be an instance of PTR. Moreover, let Φ be the set of all valuations ϕ such that ϕ satisfies $\mathcal{C}$ and ϕ is defined for precisely the parameters of $\mathcal{Q}$ and $\mathcal{P}$, and let Φ' be the set of all valuations $\phi \in \Phi$ such that $|\phi(T)| \leq \max\{1, \deg_T(\mathcal{Q}), \deg_T(\mathcal{P})\}$ for all type variables $T \in \mathrm{dom}(\phi)$. Then any maximal set Ψ of non-isomorphic valuations in Φ' is finite and the answer to the instance $\mathcal{Q}, \mathcal{P}, \mathcal{C}$ of PTR is positive if and only if $[\![\mathcal{Q}]\!]_\psi \succeq_{\mathrm{tr}} [\![\mathcal{P}]\!]_\psi$ for all valuations $\psi \in \Psi$.*

The proof of the theorem is analogous to the proof of Theorem 14 in [2].

Theorem 7 is clearly applicable to our SRS model, because $\mathcal{Q}_{Srs}$, $\mathcal{P}_{Srs} \setminus \mathcal{S}_{lock}$ and $\mathcal{C}_{Srs}$ are closed, $\mathcal{Q}_{Srs}$ has no hiding sub-LTSC and $\mathcal{C}_{Srs}$ is universal. As neither of the LTSCs has no more than two nested x-replicated sub-LTSCs such that $\delta(x) = U$ and no more than three nested y-replicated sub-LTSCs such that $\delta(y) = R$, to prove the system correct for all parameter values, it is sufficient to pick a maximal set of non-isomorphic valuations ϕ with the domain $\{U, R, <_R\}$

such that ϕ satisfies $\mathcal{C}_{Srs}$, $|\phi(U)| \leq 2$ and $|\phi(R)| \leq 3$, and then check whether the instances of the system generated by those valuations are correct.

4 Reduction Algorithm

Being aware of the structure of SRS, the representative valuations up to the cut-offs are easy to construct. There are 14 of them and they are illustrated in Fig. 1. The corresponding system and specification instances are straightforward to construct and check with the aid of an existing refinement checker.

However, in general, it would be convenient to be able to apply the theorem automatically. An obvious idea to algorithmically determine the valuations in Fig. 1 is to go through all valuations ϕ with the domain $\{U, R, <_R\}$ such that $\phi(U) = \{a_{U.1}\}, \{a_{U,1}, a_{U,2}\}$; $\phi(R) = \{a_{R,1}\}, \{a_{R,1}, a_{R,2}\}, \{a_{R,1}, a_{R,2}, a_{R,3}\}$ and $\phi(<_R)$ is a subset of $\phi(R) \times \phi(R)$. Meanwhile, we pick those valuations that satisfy $\mathcal{C}_{Srs}$ and remove isomorphs. Note that you may use any two U-atoms and any three R-atoms in the values of respectively U and R, because the model class of a VF is closed under isomorphism [2, Lemma 15] and the choice of atoms for the values of type variables is insignificant.

The algorithm sketched above is actually the one presented in [2]. However, it leaves some practically relevant questions open: namely how to generate and check all valuations and how to identify isomorphs. Moreover, although it may be necessary to generate all such valuations, for example in the case when the VF is $\top$ and there is no relation variable, often only a small number of valuations satisfy the VF and are non-isomorphic. Hence, a method for restricting a search space would be useful, too.

A traditional approach to solve classification problems like this is based on the recursive generation of a search tree [20]. An obvious idea is to start from the smallest valuation and gradually enlarge valuations as we go deeper in the tree. Moreover, as valuations that map type variables to sets of different size are necessary non-isomorphic, a search tree can be generated for each combination of type variable values separately.

Generating a search tree in the above way necessitates ordering valuations from the smallest to the largest. A natural choice for the purpose is the subvaluation order defined as follows. Let ϕ and ψ be valuations. The valuation ψ is a *subvaluation (of ϕ)* and ϕ is a *supervaluation (of ψ)*, denoted by $\psi \Subset \phi$, if ϕ and ψ have the same domain, $\psi(T) = \phi(T)$ for all type variables $T \in \text{dom}(\psi)$, $\psi(\Pi) \subseteq \phi(\Pi)$ for all relation variables $\Pi \in \text{dom}(\psi)$ and $\psi(x) = \phi(x)$ for all atom variables $x \in \text{dom}(\psi)$.

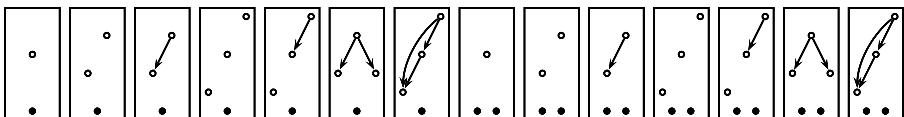


Fig. 1. Valuations to be checked for the correctness of SRS. Black dots denote users and white ones resources, and an edge turns to a resource from the proper ancestor.

We can now generate search trees using the subvaluation order $\Subset$; the root of the search tree is a valuation that maps relation variables to the empty set, and the children of a valuation ϕ are the minimal proper supervaluations ϕ' of ϕ (i.e. $\phi \Subset \phi'$, $\phi \neq \phi'$ and there is no $\phi'' \neq \phi, \phi'$ such that $\phi \Subset \phi'' \Subset \phi'$). Although all necessary valuations will be generated this way, a valuation may occur multiple times in a search tree, which leads to inefficiency. Therefore, the subvaluation order has to be strengthened.

We assume a total order $\leqslant$ in the set of atoms and type, relation and atom variables. The order is extended to tuples by generalising it to the lexicographical order, and to finite sets by considering them $\leqslant$-ordered tuples. Now, $\leqslant$ can be used to order any structures obtained from atoms, type, relation and atom variables by a recursive composition of tuples and sets. Hence, especially valuations can be ordered by $\leqslant$. Now, the root of a search tree is the same as earlier but the children of a valuation ϕ are the minimal proper supervaluations ϕ' of ϕ such that $\phi \leqslant \phi'$. This way, each valuation will be generated precisely once.

The second problem concerns detecting and removing isomorphs. This is important as it allows us to reduce the number of remaining finite-state refinement checks, which are often more expensive to perform than isomorphism rejection. A typical approach to the task is to first compute a canonical form, which is the same for all the isomorphic valuations and only them, for each valuation and then preserve only one valuation per canonical form.

To compute the canonical form of a valuation, we can exploit existing algorithms for vertex coloured graphs. That is because the vertex and edge coloured graphs can be converted to vertex coloured ones [3], and a valuation ϕ can be thought as a vertex and edge coloured graph, where the vertices and edges are respectively the atoms and the pairs of atoms in the images of, respectively, type and relation variables, the colour of a vertex a is the unique type variable $T \in \text{dom}(\phi)$ such that $a \in \phi(T)$ (plus the set of atom variables $x \in \text{dom}(\phi)$ such that $\phi(x) = a$, if there are any) and the colour of an edge (a_1, a_2) is the set of all relation variables $\Pi \in \text{dom}(\phi)$ such that $(a_1, a_2) \in \phi(\Pi)$. The canonical form of a valuation ϕ can be now defined as the canonical form of the derived vertex coloured graph and the $\leqslant$-ordered domain of ϕ as there may be relation variables that are mapped to the empty set and therefore do not colour any edge.

Note that isomorphs can be removed only after the search tree is generated. Although it would be tempting to prune the search as soon as a valuation isomorphic to a previously generated one is detected, you cannot do so, because the subtrees of isomorphic valuations are generally not isomorphic. Therefore, the pruning techniques presented in [20] cannot be applied.

However, it is allowable to prune the search tree if the VF is universal and relation-negated, and we have generated a valuation ϕ that does not satisfy the VF. That is because by the following lemma, no supervaluation of ϕ can satisfy the VF either.

Lemma 8. *Let ϕ and ψ be valuations and $\mathcal{C}$ a universal and relation-negated VF. If $\psi \Subset \phi$ and ϕ satisfies $\mathcal{C}$, then ψ satisfies $\mathcal{C}$, too.*

The lemma says that the model class of a universal relation-negated VF is downward-closed with respect to the subvaluation order. To see that it holds, first note that it is true for negated elementary VFs and (negated) always-true VFs. Therefore, it holds for all VFs obtained from these using standard Boolean connectives, i.e. all unquantified negation-normal relation-negated ones. The claim now follows from the fact that each universal relation-negated VF can be converted into a VF $\forall x_1, \ldots, x_n : \mathcal{C}'$ with the same class of models, where $\mathcal{C}'$ is unquantified, negation-normal and relation-negated.

The idea can be extended to a pruning heuristic for any universal VF $\mathcal{C}$. Let $\mathcal{C}'$ be the conjunction of all the relation-negated conjuncts of $\mathcal{C}$. Here, a *conjunct* is a maximal sub-VF that is not conjunctive. Now, $\mathcal{C}'$ is universal and relation-negated, and every valuation that satisfies $\mathcal{C}$ satisfies $\mathcal{C}'$, too. Hence, if a valuation ϕ does not satisfy $\mathcal{C}'$, then the search can be pruned because neither $\mathcal{C}'$ and hence nor $\mathcal{C}$ can be satisfied by any supervaluation of ϕ. You should note that the heuristic can only improve the search by making the search trees smaller, not enlarge them and hence impair the search.

The recursive generation of search trees with the isomorphism rejection and the pruning heuristic above is formalised as an algorithm in Fig. 2. The following property results from the fact that the algorithm implements Theorem 7.

Theorem 9. *Let $\mathcal{Q}, \mathcal{P}, \mathcal{C}$ be an instance of PTR. Then Reduce$(\mathcal{Q}, \mathcal{P}, \mathcal{C})$ returns a finite set Ψ of valuations such that the answer to the instance $\mathcal{Q}, \mathcal{P}, \mathcal{C}$ of PTR is positive if and only if $[\![\mathcal{Q}]\!]_\phi \succeq_{\mathrm{tr}} [\![\mathcal{P}]\!]_\phi$ for all $\phi \in \Psi$.*

```
procedure Reduce(Q, P, C)
   let C' be the conjunction of all the relation-negated conjuncts of C
   let Ψ := ∅
   for all valuations φ̂ defined for precisely the parameters of Q and P such that
      φ̂(T) = {a_{T,1}, ..., a_{T,k}}, where 1 ≤ k ≤ max{1, deg_T(Q), deg_T(P)}, for all
      type variables T and φ̂(Π) = ∅ for all relation variables Π in the domain
   do
      let Φ := TreeSearch(φ̂, ∅, C')
      remove valuations that do not satisfy C and isomorphs from Φ
      let Ψ := Ψ ∪ Φ
   end do
   return Ψ

subprocedure TreeSearch(φ, Φ, C')
   if φ satisfies C' then
      let Φ := Φ ∪ {φ}
      let Φ' be the set of all the minimal proper supervaluations φ' of φ s.t. φ ⩽ φ'
      for all φ' ∈ Φ' let Φ := TreeSearch(φ', Φ, C')
   end if
   return Φ
```

Fig. 2. Algorithm *Reduce* reduces the set of valuations related to an instance $\mathcal{Q}, \mathcal{P}, \mathcal{C}$ of PTR to a finite subset Ψ without changing the answer to the verification task

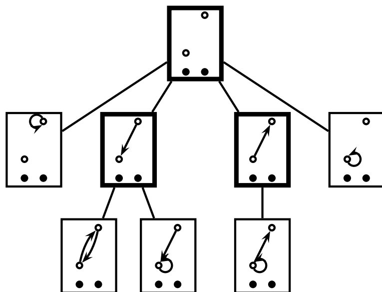


Fig. 3. The search tree for valuations with two users and two resources: the highlighted valuations satisfy $\mathcal{C}_{Srs}$, the leaves do not satisfy $\mathcal{C}'$ nor $\mathcal{C}_{Srs}$, and one child of the root is removed as an isomorph

When applied to SRS, the algorithm forms $\mathcal{C}'$ as the conjunction of the first two conjuncts of $\mathcal{C}_{Srs}$. Hence, the search can be pruned as soon as a valuation ϕ such that $\phi(<_R)$ is not irreflexive or asymmetric is encountered. After that, the algorithm generates a total of six search trees, one for each combination of (the sizes of) the values of (U, R) up to the cut-off $(2, 3)$. For example, the search tree for valuations mapping R and U to the sets of size two is shown in Fig. 3. The algorithm generates a total of 176 valuations, computes a canonical form 40 of them, and outputs 14 valuations as expected. Using the refinement checker FDR2, all the 14 instances were found to be correct, which by Theorem 9 implies that SRS works as specified for any number of users and any forest of resources.

We have implemented the algorithm in a prototype tool which uses the nauty package [3] to compute the canonical forms of valuations. With the aid of the tool and FDR2, we have verified two structurally larger examples as well: a mutual exclusion property called repeatable-read for taDOM2+ tree-locking protocol used in XML databases [21,19] and a mutual exclusion property for a ring of users that can access a shared resource in the possession of one of the two tokens circulating in the ring. In the modelling of these systems, we have applied the behavioural fixed point method as described in [19] in order to express the topologies of the systems as universal VFs.

The structure of taDOM2+ is similar to SRS except that the protocol looks different from the viewpoint of a node and its parent than from the viewpoint of a node and its other proper ancestor. Hence, two relation variables are needed to express the system from both the aspects. The same is true for the 2-token system; it looks different from the viewpoint of two users directly connected to each other than from the viewpoint of other two users. Additionally, a user that initially has both the tokens behaves in the different way, which means that three relation variables are needed to represent the system in a form that allows the algorithm to be applied.

Statistics on the execution of the tool are collected in Table 1. The isomorphism rejection seems to perform very well. The overhead of computing canonical forms, $t - t_{-iso}$, is minimal in comparison with time t_{rc} an extra refinement check

Table 1. Statistics on the performance of the algorithm: the type variables (types) used in the specification and system descriptions, the number of the relation variables with their domains (relvars), the cut-off sizes for the values of the type variables (cutoff), the size of the total search space (srch spc), the combined size of the search trees (tree sz), the number of canonical forms computed (can), the number of valuations output (out), time taken in seconds (t), time taken if the pruning heuristic is disabled (t_{all}), i.e. the whole search space is covered, time taken without the isomorphism rejection (t_{-iso}), and time taken to refinement check the largest instance up to the cut-offs (t_{rc}).

	types	relvars	cutoff	srch spc	tree sz	can	out	$t(s)$	$t_{all}(s)$	$t_{-iso}(s)$	$t_{rc}(s)$
SRS	(U,R)	$1, R \times R$	(2,3)	1060	176	40	14	0.06	0.07	0.05	91
taDOM2+	(T,N)	$2, N \times N$	(2,3)	524808	2508	110	28	0.88	33	0.87	3830
2-token	U	$3, U \times U$	4	$28 \cdot 10^{14}$	6843269	441	29	9840	–	9750	< 0.1

may take. The exception is the last case where the instances of the system are small. However, if the users in a ring would exhibit slightly more complicated behaviour, so that a refinement check would take a quarter of a second on average, then the isomorphism rejection would pay off in this case as well.

Also the pruning heuristic looks efficient. Without it, the tool performs much worse or is even unable to finish the computation, i.e. $t < t_{all}$. Hence, the heuristic seems to be useful or even vital in practical examples. However, you should note that the efficiency of the heuristic depends on the way a modeller encodes the system topology in a VF. Therefore, you ought to write the VF using as many universal relation-negated conjuncts as possible. Intuitively, it means that you should eagerly specify a topology with the aid of its complement, i.e. describe which kinds of relationships are not possible.

Despite the use of the heuristic, the running time of the algorithm is still exponential in the number of relation variables. Therefore, the introduction of a conditional LTSC is of high practical relevance, because it enables systems and specifications to be expressed with fewer relation variables than earlier. For example, the size of the total search space of the topologically simplest example, SRS, would have been in the scale of 10^{16} without the improved formalism and therefore possibly impossible to handle.

5 Conclusions

We have provided an automated cut-off method for multiparameterised verification. The method allows parameterising a system topology and guarantees successful termination on every input. We are not aware of other parameterised verification methods with these qualities. In future, we are going to extend our prototype implementation into a complete tool and equip it with the behavioural fixed-point method [5,19] so that also systems with a linear, ring or tree topology could be treated without an extra modelling step.

Acknowledgements. The research is partly funded by the Ministry of Education of Finland through Infotech Oulu Graduate School and supported by Oulu University Scholarship Foundation and the Foundation of Tauno Tönning. The author thanks Juha Kortelainen for his comments on the paper.

References

1. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
2. Siirtola, A., Kortelainen, J.: Algorithmic verification with multiple and nested parameters. In: ICFEM 2009. LNCS, vol. 5885, pp. 561–580. Springer, Heidelberg (2009)
3. McKay, B.D.: Nauty User's Guide (Version 2.4). Department of Computer Science, Australian National University (2007)
4. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
5. Valmari, A., Tienari, M.: An improved failures equivalence for finite-state systems with a reduction algorithm. In: PSTV 1991, pp. 3–18. North-Holland, Amsterdam (1991)
6. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
7. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
8. Emerson, E.A., Kahlon, V.: Model checking large-scale and parameterized resource allocation systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 251–265. Springer, Heidelberg (2002)
9. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003)
10. Emerson, E.A., Kahlon, V.: Parameterized model checking of ring-based message passing systems. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 325–339. Springer, Heidelberg (2004)
11. Li, J., Suzuki, I., Yamashita, M.: A new structural induction theorem for rings of temporal Petri nets. IEEE Trans. Softw. Eng. 20(2), 115–126 (1994)
12. Pyssysalo, T.: An induction theorem for ring protocols of processes described with predicate/transition nets. Research Report A37, Helsinki University of Technology (1996)
13. Bouajjani, A., Habermehl, P., Vojnar, T.: Verification of parametric concurrent systems with prioritised FIFO resource management. Form. Method. Syst. Des. 32(2), 129–172 (2008)
14. Delzanno, G., Raskin, J.F., Begin, L.V.: Towards the automated verification of multithreaded Java programs. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 173–187. Springer, Heidelberg (2002)
15. Bingham, J.D., Hu, A.J.: Empirically efficient verification for a class of infinite-state systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 77–92. Springer, Heidelberg (2005)

16. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
17. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Englewood Cliffs (1985)
18. Siirtola, A.: Algorithmic Multiparameterised Verification of Safety Properties. Process Algebraic Approach. PhD thesis, University of Oulu (2010)
19. Siirtola, A.: Cut-offs with network invariants. In: ACSD 2010, pp. 105–114. IEEE, Los Alamitos (2010)
20. Kaski, P., Östergård, P.R.J.: Classification Algorithms for Codes and Designs. Algorithms and Computation in Mathematics, vol. 15. Springer, Heidelberg (2006)
21. Haustein, M., Härder, T.: Optimizing lock protocols for native XML processing. Data Knowl. Eng. 65(1), 147–173 (2008)

Automating Cut-off for Multi-parameterized Systems*

Youssef Hanna, David Samuelson, Samik Basu, and Hridesh Rajan

Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, IA, USA
{ywhanna,sralmai,sbasu,hridesh}@iastate.edu

Abstract. Verifying that a parameterized system satisfies certain desired properties amounts to verifying an infinite family of the system instances. This problem is undecidable in general, and as such a number of sound and incomplete techniques have been proposed to address it. Existing techniques typically focus on parameterized systems with a single parameter, (i.e., on systems where the number of processes of exactly one type is dependent on the parameter); however, many systems in practice are multi-parameterized, where multiple parameters are used to specify the number of different types of processes in the system. In this work, we present an automatic verification technique for multi-parameterized systems, prove its soundness and show that it can be applied to systems irrespective of their communication topology. We present a prototype realization of our technique in our tool *Golok*, and demonstrate its practical applicability using a number of multi-parameterized systems.

1 Introduction

A large class of protocols described for concurrent systems, e.g., client-server protocols and multi-threaded locking protocols, do not enforce any bound on the number of processes that constitute the systems. Behavior of systems executing such protocols are modeled as parameterized systems where the parameter specifies the number of *homogeneous* processes in the system [1]. Verification of a parameterized system, therefore, amounts to verifying every instance of the system obtained by fixing the value of the parameter. In short, if $sys(n)$ is a parameterized system, where n specifies the number of homogeneous processes in the system, then verifying whether $sys(n)$ satisfies a certain desired property involves verifying that for all possible values of n, the system satisfies the property. This problem is undecidable in general [2].

Driving Problem. There is a rich body of work on parameterized system verification [3,4,5,6] that focuses on providing sound and incomplete methods to verify a *singly*-parameterized system. For a singly-parameterized system, the parameter specifies the number of exactly one type of homogeneous processes. However, in practice, there are many systems that are inherently multi-parameterized; examples include wireless sensor networks consisting of multiple types of nodes: sensors and aggregators [7] and distributed producer-consumer based systems [8] involving multiple producers

* This work has been supported in part by the US National Science Foundation under grants CNS-06-27354, CNS-07-09217, and CCF-08-46059.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 338–354, 2010.
© Springer-Verlag Berlin Heidelberg 2010

and consumers. On the one hand, in most cases, verification techniques for singly-parameterized systems are either not applicable to multi-parameterized systems, or it is not immediate how one can extend these techniques to analyze systems with multiple parameters. On the other hand, the few techniques that can indeed verify multi-parameterized systems suffer from the drawback that they require non-trivial human guidance to obtain the appropriate protocol specification (e.g. [9,10]) and/or work only for systems with certain topologies (e.g. [11]).

Consider that a multi-parameterized system with t different types of processes is described by $sys(\bar{n}_t)$ where $\bar{n}_t := n_1, n_2, \ldots, n_t$ and the parameter n_p denotes the number of processes of type p. The objective is to verify whether the system satisfies a given property for all possible valuations of each n_p in $\bar{n}_t$. This can be realized by identifying a specific instance of the system: $sys(\bar{k}_t)$ (where $\bar{k}_t := k_1, k_2, \ldots, k_t$) such that $sys(\bar{k}_t)$ satisfies the given property if and only if $sys(\bar{n}_t)$ satisfies the same, for all $\bar{n}_t \geq \bar{k}_t$ (i.e., $\forall p : n_p \geq k_p$). The parameter values in $\bar{k}_t$ corresponding to the specific instance of the system are referred to as the *cut-off*.

Our Solution. In this paper, we propose a technique, leveraging on our previous work [12], for automatically identifying such a cut-off. The technique, unlike the existing ones, is independent of both the communication topology and the property to be verified, and relies on simple input/output automata based representation of different types of processes in the system.

We consider a set of behavioral automata (introduced in [12]) to describe the input/output behavior of different types of processes in the system. The central theme of our technique is to automatically

1. compute the set of *maximal* behavior of the system (in terms of input/output) that can be induced by output action of each type of processes in the parameterized system, and
2. identify the *minimal* instance of the parameterized system that includes all such maximal behavior.

We prove that the parameter values corresponding to this minimal instance is the cut-off; more precisely, for any $\mathtt{LTL}\backslash\mathtt{X}$ (Linear Temporal Logic without "next" operator) properties which involve either actions of exactly one process or actions of two or more directly communicating processes, the instance of the parameterized system with the cut-off valuation for the parameters satisfies the property if and only if any other larger (in terms of parameter values) instance satisfies the same property.

Significant extension of [12]. While the core of the technique described in this paper is same as the one proposed and developed in [12], there are several important and non-trivial issues that are addressed in the current paper. In [12], cut-off valuation is computed for parameterized system where the parameter specifies the number of exactly one type of homogeneous process. The computed maximal behavior, therefore, is induced by one type of process. In the current paper, as there are multiple types of processes whose number is parameterized, in Step 1, it is necessary to compute the maximal behavior induced by all of them. We show that the collection containing the induced maximal behavior by *each* type of process is equivalent to the induced maximal

behavior by processes of *all* types. We further show that, it is *sufficient* to compute the induced maximal behavior for only those types of processes that are capable of making an autonomous move (without requiring external stimuli/input) from their initial states. These two conditions reduce the complexity of identifying the induced maximal behavior in Step 1 and thereby, reduce the complexity of the overall technique. Finally, for Step 2, we use a simple breadth-first strategy for incrementing parameter values to identify system instances; such strategy was not needed in [12] as the system, under consideration, was singly parameterized.

Contribution. The summary of contributions of our technique are:

- ⋄ To the best of our knowledge, we present the first automatic technique for verifying multi-parameterized systems that has the following features:
  1. the technique is applicable for verifying $\texttt{LTL}\setminus\texttt{X}$ properties over arbitrary homogeneous processes (in contrast to [11] which focuses on resource allocation systems);
  2. the technique is automatic, requires no human intervention (unlike several methods, e.g. [9], that rely on smart representation of the system being verified);
  3. the technique is independent of the communication topology (unlike [11] that works only for systems with ring topology), which, along with automation, broadens the scope of its application in practical settings.
- ⋄ We present the implementation of our technique in a tool, *Golok* and discuss several optimizations deployed to speedup the cut-off generation process. We demonstrate the robustness and scalability of our technique and implementation using different canonical multi-parameterized systems.

Organization. This paper is organized as follows. Section 2 discusses related work. Section 3 describes our technique for specifying a system using a variant of the Dining Philosophers protocol as an illustrative example. Section 4 describes how the maximal behavior induced by a process of some type p in the context of any environment is generated and shows the procedure for generating the cut-off. Proof of soundness of our technique is presented in Section 5. Section 6 describes our tool. Section 7 presents the different case studies we used to evaluate our technique and the obtained results from our tool and Section 8 offers final remarks.

2 Related Work

There exists a large body of sound and incomplete techniques for verifying parameterized systems. Solutions proposed in [3,14] reduce the problem of parameterized system verification to verification of a corresponding property-preserving finite-state abstraction, where instead of the state of each process, constraints on the number of processes at a each state are considered. Several other techniques rely on smart representation of the behavior of parameterized systems using regular grammars [4], petri-nets and graph-grammars [5]. Another class of techniques [9,10,15,16] involves identifying the invariant of a parameterized system. The invariant captures the common behavior exhibited

by all instances of the parameterized system. A property is satisfied by the parameterized system if the invariant conforms to the property. While techniques proposed in [10] apply induction to generate such invariant for singly-parameterized systems, [16] employ context-free grammars to generate the invariants for multi-parameterized systems. Most of these techniques require user guidance to obtain the grammars and/or appropriate abstraction mapping [17].

Emerson and Kahlon [11] were the first to develop a verification technique for multi-parameterized systems based on computing a cut-off. They propose solutions in the context of resource allocation systems where each homogeneous process has a specific behavior (zero or more internal transitions followed by *acquire* followed by zero or more internal transitions followed by *release*). They provide efficient methods for obtaining cut-offs when the system under consideration has a ring communication topology and the properties being considered are over adjacent processes (one process relaying a token to another).

Sun *et al.* [18] show that appropriate counter abstraction can be used to deal with state-space explosion without compromising fairness in model checking. The technique has been further applied in the context of parameterized systems by considering some pre-specified cut-off of the parameter, and any counter valuations greater than the cut-off are abstracted in the abstract model. Note that the cut-off valuation is not computed based on the model and/or the property under consideration; instead cut-off valuation is selected by the user.

Unlike these existing techniques, our technique does not rely on smart representations and/or abstractions that may require user-guidance. Our technique is fully automatic, applicable to any communication topology and is not developed in the context of any specific application domain (e.g., resource allocation).

3 Multi-parameterized System

Illustrative Example. The terminology used in this paper and the salient aspects of the proposed technique are explained using a variant of the Dining Philosophers protocol [19] (a model illustrating a classic multi-process synchronization problem). We use a variant of this protocol referred to as the Right-Left Dining Philosophers (RLDP) algorithm [11], where there are two types of philosophers: "Left" philosophers grab the left fork first and "Right" philosophers grab the right fork first. In this protocol, adjacent philosophers are of different types; therefore, the number of "Left" and "Right" philosophers is equal. Our technique is based on the notion of *behavioral automata* introduced in [12].

3.1 Processes as Behavioral Automata

Definition 1 (Behavioral Automaton). *A behavioral automaton A is a tuple (q_I, q_F, Δ, E), where q_I is the initial state, q_F is the final state, $\Delta \subseteq \{E \times \{q_I\}\} \cup \{\{q_F\} \times E\} \cup \{(q_I, q_F)\}$ is the transition relation, and E is a nonempty set of events (including the empty event ϵ). We write $q_I \rightarrow q_F$ if $(q_I, q_F) \in \Delta$, $\bullet \xrightarrow{e} q_I$ if $(e, q_I) \in \Delta$ and $q_F \xrightarrow{e} \bullet$ if $(q_F, e) \in \Delta$.*

```
1  # This diner type picks up left fork first
2  process left-diner {
3   L-START:   [init, epsilon] ->[neating, begin]
4   L-ASKL:     [neating, begin]->[waitl, askl2]
5   L-REASKL: [waitl, ltaken] ->[waitl, askl2]
6   L-FREEL-NE:[neating, askl]->[neating, lfree2]
7   L-FREEL-WL:[waitl, askl]  ->[waitl, lfree2]
8   L-BUSYL-WR:[waitr, askl]  ->[waitr, ltaken2]
9   L-BUSYL-EAT:[eat, askl]   ->[eat, ltaken2]
10  L-ASKR:     [waitl, lfree] ->[waitr, askr2]
11  L-REASKR:   [waitr, rtaken]->[waitr, askr2]
12  L-FREER-NE:[neating, askr]->[neating, rfree2]
13  L-FREER-WL: [waitl, askr] ->[ waitl, rfree2]
14  L-BUSYR-WR: [waitr, askr] ->[waitr, rtaken2]
15  L-BUSYR-EAT:[eat, askr]   ->[eat, rtaken2]
16  L-EAT: [waitr, rfree]     ->[eat, rel-forks]
17  L-EAT-DONE:[eat, rel-forks]->[neating, begin]
18 }
```

(a)

```
1  # This diner type picks up right fork first
2  process right-diner {
3   R-START: [init, epsilon]    ->[neating, begin2]
4   R-ASKR:   [neating, begin2] ->[waitr, askr]
5   R-REASKR:   [waitr, rtaken2]->[waitr, askr]
6   R-FREER-NE:[neating, askr2]->[neating, rfree]
7   R-FREER-WR:[waitr, askr2]   ->[waitr, rfree]
8   R-FREER-WL:[waitl, askr2]   ->[waitl, rfree]
9   R-BUSYR-EAT:[eat, askr2]    ->[eat, rtaken]
10  R-ASKL:   [waitr, rfree2]   ->[waitl, askl]
11  R-REASKL:[waitl, ltaken2]   ->[waitl, askl]
12  R-FREEL-NE:[neating, askl2]->[neating, lfree]
13  R-BUSYL-WL:  [waitl, askl2] ->[waitl, ltaken]
14  R-BUSYL-WR:  [waitr, askl2] ->[waitr, ltaken]
15  R-BUSYL-EAT:[eat, askl2]    ->[eat, ltaken]
16  R-EAT:  [waitl, lfree2]     ->[eat, rel-forks2]
17  R-EAT-DONE:[eat, rel-forks2]->[neating, begin2]
18 }
```

(b)

Fig. 1. Behavioral Automata for (a) “Left” Philosophers (b) “Right” Philosophers

Figures 1(a), (b) display the behavioral automata for philosophers of both types “Left” and “Right” of the RLDP protocol respectively. The statement of the form `A: [q,e]->[q',e']` denotes an automaton with $\bullet \xrightarrow{e} q$, $q \xrightarrow{\epsilon} q'$ and $q' \xrightarrow{e'} \bullet$.

A behavioral automaton describes the state in which a process can be, and what action it can perform when it is in that state. Automaton `L-ASKL` in Figure 1(a) (Line 4) presents the behavior of a philosopher of type “Left” who, while not eating (i.e. state `neating`), receives event `begin`, changes its state to `waitl` (i.e. waiting for the left fork) and sends the request for the left fork (event `askl2`). Since neighbor philosophers are of different types, the request of the left fork requested by a “Left” philosopher is received by a philosopher of type “Right”. Automaton `R-FREEL-NE` in Figure 1(b) (Line 12) models the behavior of a “Right” philosopher who receives the request for the left fork while not eating, and replies that the fork is free (event `lfree`) so that the neighbor can take it.

An automaton with ϵ input event captures the behavior of a process where, if the process is at the initial state of this automaton, it can make a move without any external stimuli. For instance, automaton `L-START` in Figure 1(a) (Line 3) states that if a philosopher is in state `init`, she can generate the event `begin` without any input events. She changes her state to `neating` after this action.

Definition 2 (Process and System Specification). *A process specification for some type p, denoted by Prot_p, is a set of behavioral automata that represents the possible actions of a process of that type. A system specification, Prot, is the union of the process specifications for all types present in the system. At least one automaton in at least one process specification in Prot must have a transition of the form $\bullet \xrightarrow{\epsilon} q$, which represents an action without input event.*

In our example, there are two process specifications; one for the “Left” philosophers Prot_l defined by the automata in Figure 1(a), and the other for the “Right” philosophers Prot_r defined by the automata in Figure 1(b), where the types “Left” and “Right” are represented by the letters l and r respectively. Both types can initiate the protocol since the specification of each one contains an automaton with transition of the form $\bullet \xrightarrow{\epsilon} q$.

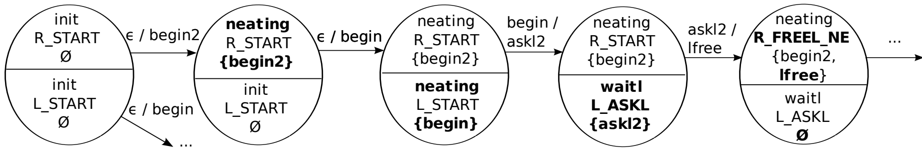


Fig. 2. Part of $sys(1_r, 1_l)$ for the RLDP Protocol

3.2 Behavior of Multi-parameterized System

Any system behavior is constrained by the topology that describes which processes in the system can directly communicate with each other.

Definition 3 (Communication Topology). *Given a system protocol specification* $\mathsf{Prot} = \bigcup_{1\le p\le t} \mathsf{Prot}_p$*, where* t *is the number of different types of processes and* $\mathsf{Prot}_p = \{A_{1_p}, \ldots, A_{l_p}\}$*, a topology is a set of tuples,* $\mathsf{Topo} \subseteq E \times (\mathcal{I} \times \mathcal{T}) \times (\mathcal{I} \times \mathcal{T})$*, where* $E = \bigcup_{1\le p\le t} \bigcup_{1\le r\le l_p} \{E_r : E_r \text{ is set of events in } A_{r_p} \in \mathsf{Prot}_p\}$*,* $\mathcal{I} \in \mathbb{N}$ *is the domain of number of processes of any type, and* $\mathcal{T}$ *is the domain of types. A tuple* $(e, i_{p1}, j_{p2}) \in \mathsf{Topo}$ *implies that output* e *from* i*-th process of type* p_1 *is consumed by the* j*-th process of type* p_2*.*

For our example, we enforce two such constraints on communication patterns: first that adjacent philosophers are of different types (therefore there is an equal number of "Left" and "Right" philosophers), and second that it is a ring topology. For instance, the topology for the system instance containing one "Left" and one "Right" philosophers is $\mathsf{Topo} = \{(\texttt{begin}, 1_l, 1_l), (\texttt{begin2}, 1_r, 1_r),\ (\texttt{askl2}, 1_l, 1_r),\ (\texttt{lfree}, 1_r, 1_l), \ldots\}$.

Definition 4 (Multi-Parameterized System). *Given a specification* Prot *with* t *different types of processes, a multi-parameterized system containing* n_p *number of processes of type* p *(*$p \in [1, t]$*) is defined as* $sys(\bar{n}_t) = (S, S_I, T, \mathsf{Topo})$*, where* $\bar{n}_t := n_1, n_2, \ldots, n_t$*,* S *is the set of states,* $S_I \subseteq S$ *is the set of initial states and* $T \subseteq S \times E \times E \times S$ *is the transition relation. A state in* S *contains* $\sum_{p=1}^{t} n_p$ *tuples of the form* $(q_{i_p}, A_{i_p}, \mathcal{E}_{i_p})$*; the tuple represents the configuration of the* i*-th process of type* p *such that* q_{i_p} *is the state of the process in the behavioral automata* A_{i_p} *and* $\mathcal{E}_{i_p}$ *denotes the set of output events from the process that have not been consumed yet.*

We use $s \overset{e/e'}{\to} s'$ *to denote* $(s, e, e', s') \in T$*.*

1. *A transition of the form* $\langle (q_{i_p}, A_{i_p}, \mathcal{E}_{i_p}), C\rangle \overset{\epsilon/e}{\to} \langle (q'_{i_p}, A_{i_p}, \mathcal{E}'_{i_p}), C\rangle \in T$*, if* $\{\bullet \overset{\epsilon}{\to} q_{i_p}, q_{i_p} \to q'_{i_p}, q'_{i_p} \overset{e}{\to} \bullet\} = \Delta \in A_{i_p} \ \wedge\ \mathcal{E}'_{i_p} = \mathcal{E}_{i_p} \cup \{e\}$.
   In the above C *represents the configurations of the remaining processes in the state.*
2. *A transition of the form* $\left\langle \begin{array}{c} (q_{i_{p1}}, A_{i_{p1}}, \mathcal{E}_{i_{p1}}) \\ (q_{j_{p2}}, A_{j_{p2}}, \mathcal{E}_{j_{p2}}) \\ C \end{array} \right\rangle \overset{e/e'}{\to} \left\langle \begin{array}{c} (q'_{i_{p1}}, A'_{i_{p1}}, \mathcal{E}'_{i_{p1}}) \\ (q_{j_{p2}}, A_{j_{p2}}, \mathcal{E}'_{j_{p2}}) \\ C \end{array} \right\rangle \in T$*, if*

$$\{\bullet \overset{e}{\to} q_{i_{p1}}, q_{i_{p1}} \to q'_{i_{p1}}, q'_{i_{p1}} \overset{e'}{\to} \bullet\} = \Delta \in A_{i_{p1}} \ \wedge \\ \mathcal{E}'_{i_{p1}} = \mathcal{E}_{i_{p1}} \cup \{e'\} \ \wedge\ \mathcal{E}_{j_{p2}} = \mathcal{E}'_{j_{p2}} \cup \{e\} \ \wedge\ (e, j_{p2}, i_{p1}) \in \mathsf{Topo}$$

Figure 2 shows part of the system instance with one philosopher of type "Right" and one of type "Left", $sys(1_r, 1_l)$. Each state (system configuration) contains two process configurations for processes 1_l, 1_r, respectively. Two possible transitions (see Rule 1 in Definition 4) can happen from the initial configuration: the transition on ϵ/`begin2` belongs to move done by philosopher 1_r and the transition on ϵ/`begin` belongs to the one by philosopher 1_l. As these moves require no external stimuli (no input event), we call these moves *autonomous moves*. The second state in the figure shows the effect of the autonomous move done by philosopher 1_r on her configuration, where her state changes and her set of output events has the produced event. The transition on `begin/askl2` in the figure illustrates a non-autonomous move (see Rule 2 in Definition 4) with intra-process communication, where the philosopher 1_l consumes the event `begin` that she has produced from her own previous autonomous move, and produces the event `askl2` as a result (to ask the left fork). In the figure, the last transition on `askl2/lfree` illustrates a non-autonomous move with inter-process communication, where the request `askl2` for the right fork produced by philosopher 1_l is received by philosopher 1_r (according to Topo). Philosopher 1_r tells her neighbor that she can take the fork by sending the event `lfree`.

4 Cut-off Computation for Multiple Parameters

In this section, we describe our technique for computing the cut-off value for a multi-parameterized system. Given the specification for all process types in the system as behavioral automata and the topology as input, our technique consists of two steps. First, it computes the maximal behavior a process of each type can induce when it autonomously produces an event to be consumed by the environment. Second, it finds a multi-parameterized system instance whose behavior exhibits all the maximal behaviors that can be induced by processes of different types (if such an instance exists). We prove that the size of this system instance is the cut-off for the multi-parameterized system. We proceed with the computation of the maximal behavior induced by a process.

4.1 Maximal Behavior Induced by a Process

Intuitively, the maximal behavior of a system induced by a process of type p is all possible sequences of input/output events that can be caused by an autonomous move done by the process. We will use π (with appropriate subscripts) to denote sequence of input/output events.

Definition 5 (Maximal Behavior induced by type p process). *Given a multi-parameterized system $sys(\bar{k}_t)$, the maximal behavior induced by a process of type $p \in [1, t]$, denoted by $\textsc{Max}_p(sys(\bar{k}_t))$, is*

$$\textsc{Max}_p(sys(\bar{k}_t)) = \{\pi_p \mid \forall i \geq 0 : \pi_p[i] = \pi[h_p^\pi(i)] \wedge \eta_0 \in S_0 \wedge \forall j \geq 0 : \eta_j \stackrel{\pi[j]}{\rightarrow} \eta_{j+1}\}$$

In the above, $h_p^\pi(i) = k$ such that

1. $\pi[k] = \epsilon/e$, $\eta_k \xrightarrow{\pi[k]} \eta_{k+1}$ *is an autonomous move of type* p *process and* $\forall j \in [h^\pi_p(0), h^\pi_p(i-1)] : \pi[h^\pi_p(j)] \neq \epsilon/e'$.
2. $\pi[k] = e_1/e_2$, $\pi[h^\pi_p(i-1)] = e/e_1$ *and* $\forall j \in [h^\pi_p(i-1), k-1] : \pi[j] \neq e_1/e'$.

For instance, for the system $sys(1_r, 1_l)$ displayed in Figure 2, the set of maximal behavior that can be induced by a philosopher of type "Left", denoted as $\textsc{Max}_l(sys(1_r, 1_l))$, is composed of all possible sequences of input/output events that can occur as a result of a "Left" philosopher that autonomously makes a move. Let $\pi_l \in \textsc{Max}_l(sys(1_r, 1_l))$. The first input/output event in such a sequence π_l belongs to a "Left" philosopher making an autonomous move to initiate the protocol by sending event `begin` (i.e. $\pi_l[0] = \epsilon/$`begin`). The second input/output event of is of the same philosopher receiving this event and sending the request for left fork (i.e. $\pi_l[1] =$ `begin/askl2`). The third input/output event belongs to a philosopher of type "Right" that responds to the request of the left fork (i.e. $\pi_l[2] =$ `askl2/lfree`), and so on. Since the event $\epsilon/$`begin2` which belongs to the move done by a "Right" philosopher is not induced by the autonomous move of the "Left" philosopher, this event does not belong to any sequence in $\textsc{Max}_l(sys(1_r, 1_l))$.

Computing the Induced Maximal Behavior. The computation proceeds by *chaining* of output from one behavior automata (present in the system specification) with the input (having the same name as the output) to another behavioral automata. For computing sequences in $\textsc{Max}_p(sys(\bar{k}_t))$, the first automata used in this chaining contains the initial state of the process of type p from where the process can make an autonomous move. We refer the result of such chaining as 1E_p and we show that 1E_p includes all possible behavior induced by the process of type p.

Definition 6 (1E_p). *Given a specification* $\mathit{Prot} = \{A_1, A_2, \ldots, A_m\}$ *with* t *different types of processes,* 1E_p *of the process type* p *is defined as a tuple* $(Q_{1\text{E}_p}, Q_{I1\text{E}_p}, \Delta_{1\text{E}_p})$*, where* $Q_{1\text{E}_p} = \{(q_I, A), (q_F, A) \mid A = (q_I, q_F, \Delta, E)\}$, $Q_{I1\text{E}_p} = \{(q_I, A) \mid A = (q_I, q_F, \Delta, E) \wedge \bullet \xrightarrow{\epsilon} q_I \in \Delta\}$*, and*

$$\Delta_{1\text{E}_p} = \left\{ (q_I, A) \xrightarrow{e_1/e_2} (q_F, A) \mid A = (q_I, q_F, \Delta, \{e_1, e_2\}), \{\bullet \xrightarrow{e_1} q_I, q_2 \xrightarrow{e_2} \bullet\} \subseteq \Delta \right\}$$
$$\bigcup \left\{ \begin{array}{r} (q_F, A) \xrightarrow{\tau} (q'_I, A') \mid A = (q_I, q_F, \Delta, E), A' = (q'_I, q'_F, \Delta', E'), \\ q_F \xrightarrow{e} \bullet \in \Delta, \bullet \xrightarrow{e} q'_I \in \Delta' \end{array} \right\}$$

Figure 3 presents a partial view of 1E_l for the "Left" philosopher processes. Automaton `L-START` is chained to automaton `L-ASKL` as their corresponding output and input events match (`begin`). Similarly, automaton `L-ASKL` is chained with automata `R-FREEL-NE` and `R-BUSYL-WR` (defined in Figure 1(b), Lines 6 and 7 respectively) due to matching output and input events (`askl2`).

Note that not all the behavioral automata of the philosophers of type "Left" in Figure 1(a) will be included in 1E_l. For instance, automaton `L-FREEL-NE` (Figure 1(a), Line 12) models the behavior of the philosopher of type "Left" that replies to a request for the left fork that comes from its neighbor (i.e., a "Right" philosopher). This

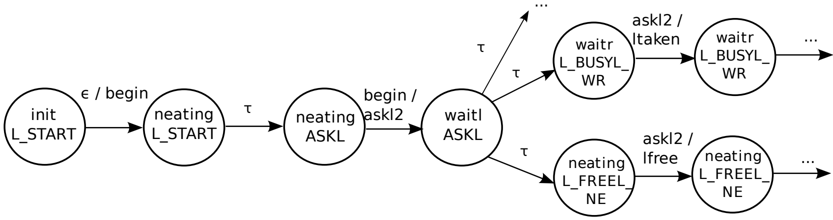


Fig. 3. Part of $1\mathrm{E}_l$

automaton will be present in the automata chain used in the construction of $1\mathrm{E}_r$ (all possible behavior induced by the autonomous output from a "Right" philosopher).

We prove that every sequence in $\mathrm{MAX}_p(sys(\bar{k}_t))$ is present as a path in $1\mathrm{E}_p$. Note that, paths in $1\mathrm{E}_p$ contains τs obtained due to chaining of automata (over same output-input event pairs). We discard these events as they are connectors between the automata that do not contribute to any action. Given a sequence of events (say π) obtained from a path in $1\mathrm{E}_p$, the corresponding sequence $\pi_{-\tau}$ is obtained by removing τ events from π as follows: $\forall i \geq 0$,

$$\pi_{-\tau}[i] = \pi[g(i)] \text{ where } g(i) = \begin{cases} 0 \text{ if } i < 0 \\ k \text{ otherwise; } g(i-1) \leq j < k : \pi[j] = \tau \wedge \pi[k] \neq \tau \end{cases} \tag{1}$$

Proceeding further, we define the set of sequences of input/output events in $1\mathrm{E}_p = (Q_{1\mathrm{E}_p}, Q_{I1\mathrm{E}_p}, \Delta_{1\mathrm{E}_p})$ as

$$\mathrm{PATH}(1\mathrm{E}_p) = \{\pi_{-\tau} \mid \zeta_0 = (q, A_x) \in Q_{I1\mathrm{E}_p} \ \wedge \ \forall i \geq 0 : \zeta_i \xrightarrow{\pi[i]} \zeta_{i+1} \in \Delta_{Q_{1\mathrm{E}_p}}\}$$

Theorem 1. *Given a protocol specification* Prot *with t different types of processes,* $\forall \bar{k}_t, \forall p \in [1, t] : \mathrm{MAX}_p(sys(\bar{k}_t)) \subseteq \mathrm{PATH}(1\mathrm{E}_p)$.

For ease of explanation and brevity of the proof, we introduce the following functions.

$$F_1(\pi, \Pi) = \{\pi' \mid \pi' \in \Pi \wedge \pi' \sqsubset \pi \wedge \nexists \pi'' \in \Pi : (\pi' \sqsubset \pi'' \sqsubset \pi) \vee (\pi'' = \pi)\} \tag{2}$$

where $\sqsubset$ denotes the strict substring relationship, i.e., $\pi' \sqsubset \pi$ implies π' is a substring of π and $\pi' \neq \pi$. The above function computes a set of substrings π' of π such that there are no other substrings of π in Π that are longer than the elements in the resultant set. We define the following function over sequences of events.

$$F_2(\pi, \pi') = e_1/e_0 \text{ such that } \pi' \sqsubset \pi \ \wedge \ \pi[|\pi'|] = e_1/e_0 \tag{3}$$

The above function identifies the event on which the sequence π diverges from π'.

Proof. Assume that $\exists \bar{k}_t, \exists p \in [1, t] : \mathrm{MAX}_p(sys(\bar{k}_t)) \not\subseteq \mathrm{PATH}(1\mathrm{E}_p)$. In other words, there exists a π such that $\pi \in \mathrm{MAX}_p(sys(\bar{k}_t))$ and $\pi \notin \mathrm{PATH}(1\mathrm{E}_p)$. From Equations 2 and 3, $F_1(\pi, \mathrm{PATH}(1\mathrm{E}_p)) = \chi$ and $\forall \pi' \in \chi : \exists e_1/e_0 : F_2(\pi, \pi') = e_1/e_0$.

There are two possible cases.

Case 1. $e_1 = \epsilon$. According Definition 5, the event e_1/e_0 is an an autonomous move of a process of type p in $sys(\bar{k}_t)$. Since the first transition of 1E_p models with an autonomous move of type p process (Definition 6), this case is not possible, i.e., our assumption that $\pi \notin \text{PATH}(1\text{E}_p)$ is false.

Case 2. $e_1 \neq \epsilon$. The event e_1/e_0 must be preceded by an input/output event of the form e_2/e_1 in order to allow for the event to happen in the first place (Definition 5). I.e., if $\pi[i] = e_1/e_0$, $\pi[i-1] = e_2/e_1$. From Equation 3, $F_2(\pi, \pi') = e_1/e_0$ and therefore, $\pi'[i-1] = e_2/e_1$ and $\pi'[i] \neq e_1/e_0$. In order for this to be possible, we need to conclude that there exists no behavioral automata that can consume e_1 and produce e_0, as construction of 1E_p proceeds by chaining the output (e_1 in this case) of one automata with the matching input of another. If no automata can take as input e_1 and produce e_0, then it is not possible to have any sequence π in $\text{MAX}_p(sys(\bar{k}_t))$ that has $\pi[i-1] = e_2/e_1$ and $\pi[i] = e_1/e_0$. This contradicts our assumption that $\pi \in \text{MAX}_p(sys(\bar{k}_t))$.

4.2 Finding the Cut-Off Value

The cut-off of parameter values for a parameterized system is such that the instance of the parameterized system at the cut-off (cut-off instance) satisfies a property if and only if all instances of the parameterized system larger than the cut-off instance satisfies the same property. We will consider two types of properties in the logic of $\texttt{LTL}\backslash\texttt{X}$:

- ◇ TYPE I PROPERTY: Property that involves the states of exactly one process. For example, if a philosopher tries to pick the left fork, she is eventually in a state where she can eat.
- ◇ TYPE II PROPERTY: Property involving two adjacent processes that directly communicate via input/output events. For example, two adjacent philosophers do not eat at the same point of time.

We will use the standard notation $[\![\varphi]\!]$ to denote the semantics of an $\texttt{LTL}\backslash\texttt{X}$ property φ; it represents the set of sequence of states that satisfy φ. A system sys satisfies φ, denoted by $sys \models \varphi$, if and only if all paths starting from all start states of the system result in a set of sequence of states such that this set is a subset of $[\![\varphi]\!]$. For details of semantics of $\texttt{LTL}$, please refer to [13].

Definition 7 (Cut-off). *Given a protocol specification* Prot *for t different types of processes and a topology* Topo*, for any $\texttt{LTL}\backslash\texttt{X}$ properties of Type I and Type II, denoted by φ, $\bar{k}_t := k_1, k_2, \ldots, k_t$ is said to be cut-off if and only if the following holds: $sys(\bar{k}_t) \models \varphi \Leftrightarrow \forall \bar{n}_t \geq \bar{k}_t : sys(\bar{n}_t) \models \varphi$ where $\bar{n}_t \geq \bar{k}_t \Leftarrow \forall p \in [1, t] : n_p \geq k_p$.*

To automatically identify the cut-off, we iteratively compute specific instances of the system under consideration and compute all possible sequences of input/output events in the system-instance. Such a set of sequence in a system-instance $sys(\bar{k}_t) = (S, S_I, T, \text{Topo})$ is defined as $\text{PATH}(sys(\bar{k}_t), S) = \{\pi_{-\tau} \mid \eta_0 = s \in S \ \wedge \ \forall i \geq 0 : \eta_i \stackrel{\pi[i]}{\rightarrow} \eta_{i+1} \in T\}$. We prove that $\bar{k}_t$ is the cut-off if $\forall p \in [1, t] : \text{PATH}(1\text{E}_p) \subseteq \text{PATH}(sys(\bar{k}_t), S^{\bar{k}_t})$ where $S^{\bar{k}_t}$ denotes the set of states in $sys(\bar{k}_t)$. Procedure CutOff presents our automatic method for obtaining the cut-off.

Procedure CutOff (Prot, t, Topo, initial system config)

Construct initial $sys(\bar{k}_t)$ from initial system config and Topo
for all $p \in [1, t]$ Compute $1\mathrm{E}_p$ from Prot **do**
while $\textsc{Path}(1\mathrm{E}_p) \not\subseteq \textsc{Path}(sys(\bar{k}_t), S^{\bar{k}_t})$ Increase $\bar{k}_t$ in a breadth-first manner
end while
end for
return $\bar{k}_t$;

---

5 Proof of Soundness

We proceed by introducing definitions and propositions that will be used to prove the soundness of Procedure CutOff.

Definition 8 (Projection on processes). *Given a multi-parameterized system $sys(\bar{k}_t) = (S, S_I, T, \textsf{Topo})$ with t different types of processes and a set $R \subseteq \{i_p \mid i \in [1, k_p] \;\wedge\; p \in [1, t]\}$, the projected behavior w.r.t. R is denoted by $sys(\bar{k}_t)\downarrow R = (S, S_I, T\downarrow R, \textsf{Topo})$, such that labels of all transitions that do not directly involve moves of process $i'_{p'} \in R$ are renamed to τ. We will use $\pi\downarrow R$ to denote projection of a sequence of events on R.*

For the example of the RLDP protocol, in the projected system $sys(1_r, 1_l)\downarrow\{1_l\}$, both the transitions labeled as $\epsilon/\texttt{begin}$ and the transition labeled as $\texttt{begin}/\texttt{askl2}$ remain the same while all other transitions in the Figure 2 are substituted with τ transitions.

Proposition 1. *For any multi-parameterized system $sys(\bar{k}_t)$ with t different types of processes, the following holds for all properties φ (in the logic of $\texttt{LTL}\setminus\texttt{X}$) defined over states of processes whose indices belong to $R = \{i_p \mid i \in [1, k_p] \wedge p \in [1, t]\}$: $sys(\bar{k}_t) \models \varphi \;\Leftrightarrow\; sys(\bar{k}_t)\downarrow R \models \varphi$.*

Proposition 2. *Let Φ be the set of all properties (in the logic of $\texttt{LTL}\setminus\texttt{X}$) defined over states of processes whose indices belong to $R = \{i_p \mid i \in [1, k_p] \wedge p \in [1, t]\}$. The following holds for any two instances of multi-parameterized systems, $sys(\bar{k}_t)$ and $sys(\bar{k}'_t)$.*

$$\forall \varphi \in \Phi : \big(sys(\bar{k}_t) \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t) \models \varphi\big) \Rightarrow \textsc{Path}(sys(\bar{k}_t)\downarrow R, S_I^{\bar{k}_t}) = \textsc{Path}(sys(\bar{k}'_t)\downarrow R, S_I^{\bar{k}'_t})$$

In the above, $S_I^{\bar{k}_t}$ and $S_I^{\bar{k}'_t}$ are the initial state-sets of $sys(\bar{k}_t)$ and $sys(\bar{k}'_t)$, respectively.

Proof. From Proposition 1, we conclude

$$\forall \varphi \in \Phi : \big(sys(\bar{k}_t) \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t) \models \varphi\big) \Rightarrow \big(sys(\bar{k}_t)\downarrow R \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t)\downarrow R \models \varphi\big)$$

If π denotes a path in a system over sequence of input/output actions, we denote the corresponding sequence of states in the path by $\texttt{seq}(\pi)$. Therefore,

$$\forall \varphi \in \Phi : \big(sys(\bar{k}_t){\downarrow}R \models \varphi \;\Leftrightarrow\; sys(\bar{k}'_t){\downarrow}R \models \varphi\big) \;\Rightarrow$$
$$\forall \pi \in \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) : \exists \pi' \in \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t}) : \mathtt{seq}(\pi) = \mathtt{seq}(\pi')$$
$$\bigwedge$$
$$\forall \pi' \in \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t}) : \exists \pi \in \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) : \mathtt{seq}(\pi') = \mathtt{seq}(\pi)$$
$$\Rightarrow\; \textsc{Path}(sys(\bar{k}_t){\downarrow}R, S_I^{\bar{k}_t}) = \textsc{Path}(sys(\bar{k}'_t){\downarrow}R, S_I^{\bar{k}'_t})$$

Proposition 3. *For any parameterized system with t types of processes,*

$$\forall \bar{n}_t \geq \bar{k}_t : \textsc{Path}(sys(\bar{k}_t), S_I^{\bar{k}_t}) \subseteq \textsc{Path}(sys(\bar{n}_t), S_I^{\bar{n}_t})$$
$$\forall p \in [1,t] : \textsc{Path}(1\mathrm{E}_p) \subseteq \textsc{Path}(sys(\bar{k}_t), S^{\bar{k}_t}) \Rightarrow\; \textsc{Path}(1\mathrm{E}_p) \subseteq \textsc{Path}(sys(\bar{n}_t), S^{\bar{n}_t})$$

where $S^{\bar{k}_t}$, $S_I^{\bar{k}_t}$ and $S^{\bar{n}_t}$, $S_I^{\bar{n}_t}$ are the sets of states and initial states of $sys(\bar{k}_t)$ and $sys(\bar{n}_t)$ respectively.

Theorem 2. *Given a parameterized system with t different types of processes each defined using a set of behavioral automata* Prot*, the following holds for all Type I and II properties φ in the logic of $\mathtt{LTL}\backslash\mathtt{X}$*

$$\forall p \in [1,t] : \textsc{Path}(1\mathrm{E}_p) \subseteq \textsc{Path}(sys(\bar{k}_t), S^{\bar{k}_t}) \Rightarrow\; \big(sys(\bar{k}_t) \models \varphi \Leftrightarrow sys(\bar{n}_t) \models \varphi\big)$$

where $\bar{n}_t = n_1, n_2, \ldots, n_t$, $\bar{k}_t = k_1, k_2, \ldots, k_t$, $S^{\bar{k}_t}$ is the set of states in $sys(\bar{k}_t)$, and $S_I^{\bar{k}_t}$ and $S_I^{\bar{n}_t}$ are initial state-sets of $sys(\bar{k}_t)$ and $sys(\bar{n}_t)$ respectively.

Proof. Due to space constraints, we provide a proof sketch for the theorem. The full proof is available at `http://www.cs.iastate.edu/~slede/golok/`.

Using Propositions 1, 2 and 3, it is required to prove that $\forall p \in [1,t], \forall i \leq n_p, \exists j \leq k_p$, $\textsc{Path}(sys(\bar{n}_t){\downarrow}\{i_p\}, S_I^{\bar{n}_t}) = \textsc{Path}(sys(\bar{k}_t){\downarrow}\{j_p\}, S_I^{\bar{k}_t})$.

Assume that there exists a sequence π in $\textsc{Path}(sys(\bar{n}_t){\downarrow}\{i_p\}, S_I^{\bar{n}_t})$ that is not present in $\textsc{Path}(sys(\bar{k}_t){\downarrow}\{j_p\}, S_I^{\bar{k}_t})$. This implies that for every path π' in $\textsc{Path}(sys(\bar{k}_t){\downarrow}\{j_p\}, S_I^{\bar{k}_t})$, there exists an input/output event (e_1/e_0) such that e_1/e_0 is present in π and absent in π'. If e_1 is equal to ϵ, then it can be immediately shown that $\textsc{Path}(sys(\bar{k}_t){\downarrow}\{j_p\}, S_I^{\bar{k}_t}) \not\subseteq \textsc{Path}(1\mathrm{E}_p)$ as ϵ/e_0 is an autonomous move. This forms the base case of the proof (contradiction of our assumption above). If e_1 is not equal to ϵ then there must be some event e_2/e_1 preceding e_1/e_0 in the path π and such ordering of events is absent in π'. In this case, it can be shown that π' diverges from π on event e_2/e_1. The proof of the theorem (i.e., contradiction of our assumption) can be realized by proceeding inductively (on the length of the diverging point between π and π') and eventually reaching the base case.

Theorem 3 (Soundness). *If Procedure* CutOff *terminates, the return $\bar{k}_t$ is the cut-off as per the Definition 7.*

Proof. Follows from Theorem 2.

```
1  process left-diner { ... }
2  process right-diner { ... }
3
4  topology {
5
6   connectivity {
7    left-diner 0 -- right-diner 0
8   }
9
10  additionrule add-two {
11   create: left-diner x
12   create: right-diner y
13   require: right-diner z -- left-diner 0
14   remove: var z -- left-diner 0
15   add: var z -- var x
16   add: var x -- var y
17   add: var y -- left-diner 0
18  }
19  msgs {
20   (left-diner, begin, self)
21   (left-diner, askl, rpeer)
22   (right-diner, askr2, lpeer) ...
23  }
24 }
25
26 initialconfig {  }
```

Fig. 4. Input file for the RLDP protocol

6 Golok: A Tool to Find Cut-off

We have implemented our technique in a tool, *Golok*[1]. It is written in Scheme [20] in ~4K lines of code. We now describe the input language to Golok using the RLDP example and describe the several optimizations we implemented in our tool.

6.1 Front End: Input Language of Golok

The input file containing the specification for the RLDP protocol is displayed in Figure 4. The specification has three main components: (a) the process specification, (b) the topology specification and (c) the initial configuration specification.

Process Specification. The process specifications (lines 1, 2) contain the behavioral automata for every process type as described in Section 3 (Figures 1(a) and (b)).

Topology Specification. The topology specification serves to restrict communication patterns between processes. It is defined using the keyword **`topology`** (lines 4 - 25) and is composed of three parts. The first part of the topology specification (lines 6-8) specifies the topology of the initial system instance. In RLDP, the initial system instance has one philosopher of each type (processes are zero-indexed). The second part of the topology specification (`additionrule` lines 10-18) is the addition rules that ensure that newly generated system instances follow the communication topology of the protocol. For RLDP, the addition rule `add-two` (lines 10-18) states that any new system instances will create two new philosophers of different types (lines 11-12) and that they linked to other processes to preserve that neighbors are of different types (lines 13-17). The final part of the topology specification (lines 19-23) is responsible for specifying the direction of the flow of events between processes, where every tuple (d, e, s) describes the event to be received e, the type of the recipient process d and the index of the sender process s. There are four choices for s: `self` (message sent and received by the same process), `rpeer` (message sent by the right neighbor of d) `lpeer` (message sent by the left neighbor of d) and `peer` (message sent by a neighbor of d)[2].

Initial Configuration Specification. The initial configuration is explicitly specified if any process needs to start from a different automaton other than the first automaton in its process specification.

[1] A cutting tool typically used in Indonesia and the Philippines.
[2] `lpeer`, `rpeer` used for ring topology, `peer` for other topologies.

6.2 System Instance Generator/Checker

The System Instance Generator/Checker (SIGC) is the main module of Golok. The goal of SIGC is to construct a system instance $sys(\bar{k}_t)$ (see Procedure CutOff) and check whether for all $p \in [1, t]$, $\text{PATH}(1\text{E}_p) \subseteq \text{PATH}(sys(\bar{k}_t), S^{\bar{k}_t})$. The main challenge in implementing SIGC is to reduce the computational cost involved in checking for path inclusion by considering all possible paths from all states. We describe several optimizations we have implemented in Golok to help reduce the computational cost.

Simulation-base cut-off computation. Simulation relation [21] identifies pairs of states in a transition system such that one element of the pair simulates all possible behavior (in terms of sequence and branching of transitions) of the other. It is a stronger relation than language inclusion. Furthermore, computing simulation relation is linear to the state-space of the transition system as opposed to computing language inclusion which is exponential to the state-space. As a result, it is computationally efficient to use simulation rather than language inclusion. The results of Theorem 3 still holds. Given a protocol specification Prot and a multi-parameterized system $sys(\bar{k}_t)$ with t different types of processes, a state r in $sys(\bar{k}_t)$ is said to simulate a state s in 1E_p, denoted as $s \prec r$, if the following holds:

$$\forall e/e', s' : s \overset{\tau^* e/e'}{\rightarrow} s' \in 1\text{E}_p \Rightarrow \exists r' : r \overset{\tau^* e/e'}{\rightarrow} r' \in sys(\bar{k}_t) \land s' \prec r'$$

In the above, $\tau^* e/e'$ represents zero or more τ transitions followed by an e/e' transition. We say that 1E_p is simulated by $sys(\bar{k}_t)$ if and only if there exists a state r in $sys(\bar{k}_t)$ such that for all start states s in 1E_p, $s \prec r$.

Simulation based cut-off computation may lead to additional challenges. For certain systems where there exists a cut-off that can be identified using path inclusion, a stronger requirement for cut-off based on simulation may fail to obtain such a cut-off. From our experimental results, we have realized that such a problem exists when in addition to parameterized components, the system also contains non-parameterized components (ones whose number is pre-specified and fixed). For instance, in the bounded-buffer protocol, there exists only one buffer for all instances of the systems. Similarly, for the singly-parameterized spin lock, there is only one object in any system instance. The problem of using simulation for such systems can be alleviated by projecting out any actions that result from the non-parameterized components.

Reducing the number of Simulation Checks. As the size of a system instance could be prohibitively large, performing a simulation check on every state to verify if it simulates 1E_p can still be expensive. To reduce the number of simulation checks, we construct the system instances on-the-fly (i.e. states are generated when needed), perform partial-order reduction ([22]) to ensure re-use of intermediate simulation checking results. Furthermore, for every system configuration s in $sys(\bar{k}_t)$, the following constant-time check is done before performing a simulation check. If the system configuration s does not have any process that is able to make an autonomous move (a move that does not require any external stimuli), this system configuration s is never expanded. The reason is that, since the first transition in any 1E_p must come from an autonomous move, then it is not possible that a system configuration s where no process is able to make an autonomous move is the configuration that simulates 1E_p for any type p.

Table 1. Experimental results of our tool *Golok* compared to existing techniques

Protocol	Topology	Process Types: # of types	Process Types: # of Params	Existing Work: References	Existing Work: Known Cut-off	Our Technique: Computed Cut-off ($\bar{k}_t$)	Our Technique: Time (sec)	Our Technique: Explored States	Our Technique: States in $sys(\bar{k}_t)$	Our Technique: %gain
Dining Philosophers [11,19]	Ring	1	1	[11]	4	**3**	0.54	33	510	93.53
		2	(r, l) 2	[11]	$2_r, 2_l$	$3_r, 3_l$	4.84	7,524	268,536	97.20
Bounded-Buffer [23]	Star	3	(p, c) 2	X†	X	$2_p, 1_c$	1.10	37	269	86.25
Spin Lock [24]	Star	2	(t) 1	[26]	3	$\mathbf{2}_t$	0.62	13	84	84.50
	Multi-star*	2	(t, o) 2	X	X	$2_t, 2_o$‡	0.52	15	243	93.80
DME [25]	Ring	1	(fc) 1	[27]	4	$\mathbf{2}_{fc}$	0.51	5	7	28.58
		2	(f, c) 2	X	X	$1_f, 1_c$	0.51	4	7	42.86

*Multi-star: All processes of different types are connected; †: To the best of our knowledge, no known results exist.
‡: Golok produced same cut-off value for different sizes of the buffer, displayed performance results are for the system with buffer of size 1.
r: right philosopher, l: left philosopher; p: producer, c: consumer; t: thread, o: object; fc: dme node, f: forward dme node, c: critical dme node.

7 Case Studies

Besides the RLDP protocol, we ran Golok on three other multi-parameterized systems with different communication topologies to validate our technique: the Bounded Buffer protocol [23], a variant of the Spin Lock protocol [24] and a variant of the Distributed Mutual Exclusion Protocol [25]. All examples along with the tool, Golok, are available at `http://www.cs.iastate.edu/~slede/golok/`. All experiments were run on a single core Pentium 4, 2.53 Ghz with 2 GB of RAM. Table 1 summarizes the experimental results. First four columns presents the parameterized systems and their various features: topology, number of process types in the system and number of types of the process that are parameterized. The rest of the table provides typical solutions obtained for some of the examples from the most relevant existing work and compared it with the results obtained from our tool, Golok. To the best of our knowledge, none of the existing techniques provide with a viable tool that can be used in practice. As a result, we only provide execution time information for our technique.

The table shows that while parameterized systems with different communication topology are handled by different techniques (developed primarily for the topology under consideration), our technique is applicable uniformly to all parameterized systems (both singly- and multi-parameterized) with different communication topologies. Note that, in some cases, Golok has identified a smaller cut-off value compared to the ones known in the existing work (shown in bold font). This can be attributed primarily to the fact that existing techniques for cut-off identification are independent of the system behavior (only topology dependent, e.g., [27]) or rely on abstractions that are sufficient but not necessary (e.g., [26]). As our technique is system dependent, Golok may compute different cut-off values for different systems with the same topology.

The table also shows impact of the optimizations in our technique. For instance, the last column shows the proportion of states that are not explored in the cut-off instance of the parameterized system while verifying that the instance simulates the 1E for all types of processes that are parameterized.

8 Summary and Conclusion

We have presented a technique for generating cut-off values for each of the parameters of a multi-parameterized system, proved its soundness, implemented the technique in a tool, and demonstrated its applicability to a number of canonical case studies. As Golok provides an automated realization of our method, the tool can be effectively used even for cases where the system is non-parameterized. For example, consider that the objective is to verify RLDP protocol with N pairs of "Left" and "Right" philosophers such that N is prohibitively large and as a result, standard model checking tools fail to provide any result due to state-space explosion. In such cases, a parameterized version of the system can be considered in Golok and if a cut-off is returned (e.g., $3_r, 3_l$ for RLDP protocol) then model checking the system instance with this cut-off is equivalent to model checking the much larger system instance contains N pairs of philosophers.

Future work includes extending the expressive power of behavioral automata representation, associated formalisms, techniques and Golok to allow for specification of parameterized system whose behavior is constrained by the valuations of messages being exchanged and to allow broadcast.

References

1. Manna, Z., Pnueli, A.: An exercise in the verification of multi-process programs. Beauty is our business: a birthday salute to Edsger W. Dijkstra, pp. 289–301 (1990)
2. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
3. Clarke, E.M., Talupur, M., Veith, H.: Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
4. Bouajjani, A., Jonsson, B., Nilsson, M., Touili, T.: Regular model checking. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 403–418. Springer, Heidelberg (2000)
5. Baldan, P., Corradini, A., König, B.: A framework for the verification of infinite-state graph transformation systems. Inf. Comput. 206(7), 869–907 (2008)
6. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
7. Przydatek, B., Song, D., Perrig, A.: Sia: secure information aggregation in sensor networks. In: SenSys. (2003)
8. Byrd, G., Flynn, M.: Producer-consumer communication in distributed shared memory multi-processors. Proceedings of the IEEE 87(3), 456–466 (1999)
9. Marelly, R., Grumberg, O.: Gormel - grammar oriented model checker. Technical Report 697, The Technion (1992)
10. Arons, T., Pnueli, A., Ruah, S., Xu, J., Zuck, L.D.: Parameterized verification with automatically computed inductive assertions. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 221–234. Springer, Heidelberg (2001)
11. Emerson, E.A., Kahlon, V.: Model checking large-scale and parameterized resource allocation systems. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 251–265. Springer, Heidelberg (2002)
12. Hanna, Y., Basu, S., Rajan, H.: Behavioral automata composition for automatic topology independent verification of parameterized systems. In: ESEC/FSE 2009 (August 2009)

13. Emerson, E.A.: Temporal and modal logic, pp. 995–1072 (1990)
14. Yavuz-Kahveci, T., Bultan, T.: Verification of parameterized hierarchical state machines using action language verifier. In: MEMOCODE 2005, pp. 79–88 (2005)
15. Roychoudhury, A., Ramakrishnan, I.V.: Inductively verifying invariant properties of parameterized systems. Automated Software Engg. 11(2), 101–139 (2004)
16. Clarke, E.M., Grumberg, O., Jha, S.: Verifying parameterized networks using abstraction and regular languages. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 395–407. Springer, Heidelberg (1995)
17. Zuck, L.D., Pnueli, A.: Model checking and abstraction to the aid of parameterized systems (a survey). Computer Languages, Systems & Structures 30(3-4), 139–169 (2004)
18. Sun, J., Liu, Y., Roychoudhury, A., Liu, S., Dong, J.S.: Fair model checking with process counter abstraction. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 123–139. Springer, Heidelberg (2009)
19. Dijkstra, E.: Two starvation free solutions to a general exclusion problem. EWD 625, Plataanstraat 5, 5671 AL Neunen, The Netherlands
20. Abelson, H., et al.: Revised report on the algorithmic language scheme. Higher Order Symbol. Comput. 11(1), 7–105 (1998)
21. Milner, R.: A Calculus of Communicating Systems. Springer, Heidelberg (1982)
22. Mazurkiewicz, A.W.: Basic notions of trace theory. In: de Bakker, J.W., de Roever, W.-P., Rozenberg, G. (eds.) Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. LNCS, vol. 354, pp. 285–363. Springer, Heidelberg (1989)
23. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. Wiley, Chichester (2004)
24. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 1(1), 6–16 (1990)
25. Wolper, P., Lovinfosse, V.: Verifying properties of large sets of processes with network invariants. In: Workshop on Automatic Verification Methods for Finite State Systems, pp. 68–80 (1990)
26. Basu, S., Ramakrishnan, C.R.: Compositional analysis for verification of parameterized systems. Theor. Comput. Sci. 354(2), 211–229 (2006)
27. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL, pp. 85–94 (1995)

Method for Formal Verification of Soft-Error Tolerance Mechanisms in Pipelined Microprocessors*

Miroslav N. Velev and Ping Gao

Aries Design Automation
miroslav.velev@aries-da.com
http://www.miroslav-velev.com

Abstract. We present techniques for design and formal verification of both safety and liveness of pipelined/superscalar/VLIW processors with built-in mechanisms for soft-error tolerance. The formal verification is done with the highly automatic method of Correspondence Checking by exploiting the property of Positive Equality and efficient translations of the correctness conditions to equivalent Boolean formulas that are evaluated with SAT solvers. Soft errors are caused by radiation and cross talk between devices or wires on the chip, and will become increasingly frequent with the decreasing transistor sizes in future technologies. Soft errors can cause catastrophic failures in safety-critical applications, such as space, avionics, weapons systems, automotive, and medical devices. Thus, the need to design and efficiently formally verify pipelined microprocessors with mechanisms for soft-error tolerance.

1 Introduction

We present a method for design at a high level of abstraction and formal verification of safety and liveness of pipelined/superscalar/VLIW processors with built-in mechanisms for soft-error tolerance. The formal verification is done with the highly automatic method of Correspondence Checking by exploiting the property of Positive Equality and efficient translations of the correctness conditions to equivalent Boolean formulas that are evaluated with Boolean Satisfiability (SAT) solvers, thus allowing us to benefit from the constant stream of innovations in the extremely active research field of SAT.

In the processors that we formally verify, the radiation-hardening is done at the microarchitectural level by using the RazorII mechanism [12], where flip-flops with a special design are used to detect timing errors—caused by variations in the voltage, frequency, operating temperature, manufacturing process, as well as aging of the chip, and radiation—such that these errors are corrected by re-executing the corresponding instruction with the replay mechanism for correcting wrong speculations in the processor, as already implemented in many processors with branch prediction, or data-value prediction [47]. Traditionally, semiconductor companies have reduced the probability for such timing errors by introducing sufficient safety margins, such as a higher supply voltage, resulting in increased power consumption. The RazorII

* This research was partially supported by NASA under contract NNX10CC60P.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 355–370, 2010.
© Springer-Verlag Berlin Heidelberg 2010

mechanism allows a processor to self-monitor, self-analyze, and self-heal after timing errors, regardless of their cause—and thus makes it possible to eliminate the safety margins by operating the processor at a lower supply voltage at the point of first failure (PoFF) of a die for a given frequency, resulting in significant energy savings.

The RazorII mechanism was developed recently [12] as joint research between the University of Michigan and ARM. It represents a state-of-the-art mechanism for correcting not only radiation-induced errors, but also timing errors due to a wide variety of causes. Thus, RazorII is a more advanced approach for radiation-hardening of microprocessors, compared with the techniques at the transistor and logic level used in the radiation hardened flight-control computer RAD750 (e.g., see p. 6 of [11]). Intel have adapted a variant of RazorII [28], and other authors have proposed similar schemes [6, 25, 26]. Formally verifying processors with RazorII will be crucial for safety critical applications, including those to be used in space, avionics, weapons systems, automotive, and medical devices.

Every time the design of computer systems was shifted to a higher level of abstraction, productivity increased. The logic of Equality with Uninterpreted Functions and Memories (EUFM) [9] allows us to abstract functional units and memories, while completely modeling the control of a processor. In earlier work on applying EUFM to formal verification of pipelined and superscalar processors, some simple restrictions [31, 32] were imposed on the modeling style for defining processors, resulting in correctness formulas where most of the terms (abstracted word-level values) appear only in positive equations (equality comparisons) that are called *p-equations*. Such terms, called *p-terms* (for positive terms), can be treated as distinct constants [7], thus significantly pruning the solution space, and resulting in orders of magnitude speedup of the formal verification; this property is called *Positive Equality*. On the other hand, equations that appear in negative polarity, or in both positive and negative polarity, are called *g-equations* (for general equations), and their arguments *g-terms*. G-equations can be either *true* or *false*, and can be encoded with Boolean variables [13, 22, 37, 48] by accounting for the property of transitivity of equality [8], when translating an EUFM correctness formula to an equivalent Boolean formula. The modeling restrictions, together with techniques to model multicycle functional units, exceptions, and branch prediction [33], allowed an earlier version of our tool flow [46] to be used to formally verify a model of the M•CORE processor at Motorola [17], and detect three bugs, as well as corner cases that were not fully implemented.

Our tool flow consists of: 1) the term-level symbolic simulator TLSim [46], used to symbolically simulate the high-level implementation and specification processors, and produce an EUFM correctness formula; 2) the decision procedure EVC [46] that exploits Positive Equality and other optimizations to translate the EUFM correctness formula to a satisfiability-equivalent Boolean formula; and 3) an efficient SAT-solver.

Recent dramatic improvements in SAT-solvers [14, 19, 24]—see [18, 36] for comparative studies—significantly sped up the solving of Boolean formulas generated in formal verification of microprocessors. However, as found in [36], the new efficient SAT-solvers would not have scaled for solving these Boolean formulas if not for the property of Positive Equality that results in at least 5 orders of magnitude speedup when formally verifying complex dual-issue superscalar processors. Efficient translations to CNF [38, 39, 41–44], exploiting the special structure of EUFM formulas produced with the modeling restrictions, resulted in additional speedup of 2 orders of magnitude.

This paper is structured as follows. Sect. 2 introduces the background of formal verification of pipelined processors by exploiting Positive Equality, and the RazorII mechanism for timing-error tolerance. Sect. 3 describes our method for abstracting any mechanism for detection and correction of soft errors or timing errors, so that pipelined processors with it can be formally verified for both safety and liveness. Sect. 4 presents experimental results, Sect. 5 discusses related work, and Sect. 6 concludes the paper.

2 Background

2.1 Formal Verification of Pipelined Processors

The formal verification is done by correspondence checking—comparing a pipelined implementation against a non-pipelined specification, using controlled flushing [10] to automatically compute an abstraction function, Abs, that maps an implementation state to an equivalent specification state. The safety property (see Fig. 1) is expressed as a formula in the logic of EUFM, and checks that one step of the implementation corresponds to between 0 and k steps of the specification, where k is the issue width of the implementation. FImpl is the transition function of the implementation, and FSpec is the transition function of the specification. We will refer to the sequence of first applying Abs and then FSpec as the specification side of the diagram in Fig. 1, and to the sequence of first applying FImpl and then Abs as the implementation *side*.

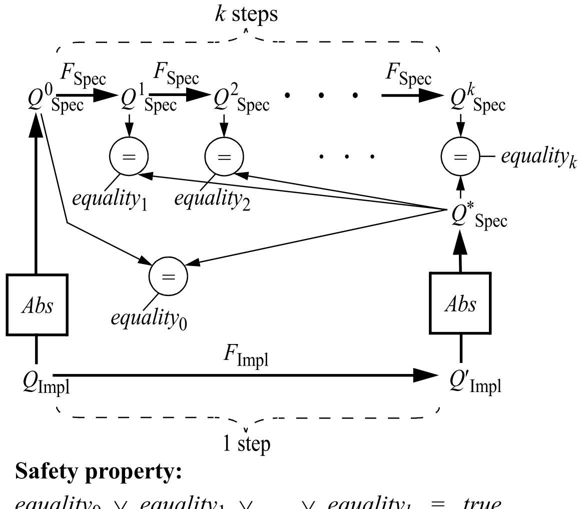


Fig. 1. The safety correctness property for an implementation processor with issue width k: one step of the implementation should correspond to between 0 and k steps of the specification, when the implementation starts from an arbitrary initial state Q_{Impl} that is possibly restricted by a set of invariant constraints.

The safety property is the inductive step of a proof by induction, since the initial implementation state, Q_{Impl}, is completely arbitrary. If the implementation is correct for all transitions that can be made for one step from an arbitrary initial state, then the implementation will be correct for one step from the next implementation state, Q'_{Impl}, since that state will be a special case of an arbitrary state as used for the initial state, and so on for any number of steps. For some processors, e.g., where the control logic is optimized by using unreachable states as don't-care conditions, we might have to impose a set of *invariant constraints* for the initial state, Q_{Impl}, in order to exclude unreachable states. Then, we need to prove that those constraints will be satisfied in the implementation state after one step, Q'_{Impl}, so that the correctness will hold by induction for that state, and so on for all subsequent states. The reader is referred to [1, 2] for a discussion of correctness criteria, and to [49] for a debugging methodology for correspondence checking.

The syntax of EUFM [9] includes *terms* and *formulas*. Terms are used to abstract word-level values of data, register identifiers, memory addresses, as well as the entire states of memories. A term can be an Uninterpreted Function (UF) applied to a list of argument terms, a term variable, or an *ITE* operator selecting between two argument terms based on a controlling formula, such that *ITE*(*formula*, $term_1$, $term_2$) will evaluate to $term_1$ when *formula* = *true*, and to $term_2$ when *formula* = *false*. The syntax for terms can be extended to model memories by means of functions *read* and *write* [9, 35]. Formulas are used to model the control path of a processor, as well as to express a correctness condition. A formula can be an Uninterpreted Predicate (UP) applied to a list of argument terms, a Boolean variable, an *ITE* operator selecting between two argument formulas based on a controlling formula, or an equation (equality comparison) of two terms. Formulas can be negated and combined with Boolean connectives. We will refer to both terms and formulas as *expressions*. If we exclude functions *read* and *write* from the syntax of EUFM, we will obtain the logic of Equality with Uninterpreted Functions (EUF).

UFs and UPs are used to abstract the implementation details of functional units by replacing them with "black boxes" that satisfy no particular properties other than that of *functional consistency*. Namely, that equal combinations of values to the inputs of the UF (or UP) produce equal output values. Thus, we will prove a more general problem—that the processor is correct for any functionally consistent implementation of its functional units. However, this more general problem is easier to prove.

Function *read* takes two argument terms serving as memory state and address, respectively, and returns a term for the data at that address in the given memory. Function *write* takes three argument terms serving as memory state, address, and data, and returns a term for the new memory state. Functions *read* and *write* satisfy the *forwarding property of the memory semantics*: *read*(*write*(*mem*, *waddr*, *wdata*), *raddr*) is equivalent to *ITE*((*raddr* = *waddr*), *wdata*, *read*(*mem*, *raddr*)).

We classify the equations that appear negated as *g-equations* (for general equations), and as *p-equations* (for positive equations) otherwise. We classify all terms that appear as arguments of g-equations as *g-terms* (for general terms), and as *p-terms* (for positive terms) otherwise. We classify all applications of a given UF as g-terms if at least one application of that UF appears as a g-term, and as p-terms otherwise.

In [31, 32], the style for modeling high-level processors was restricted in order to increase the terms that appear only in positive equations or as arguments to UFs and

UPs, and reduce the terms that appear in both positive and negated equations. First, equations between data operands, where the result appears in both positive and negated polarity—e.g., determining whether to take a branch-on-equal instruction—are abstracted with a new UP in both the implementation and the specification. Second, the Data Memory is abstracted with a conservative model, where the interpreted functions *read* and *write* are replaced with new UFs, *DMem_read* and *DMem_write*, respectively, that do not satisfy the forwarding property. This property is not needed, if both the implementation and the specification execute the same sequence of operations that are not stalled based on equations between addresses for that memory [35].

The property of functional consistency of UFs and UPs can be enforced by Ackermann constraints [3], or by nested *ITE*s [30]. The Ackermann scheme replaces each UF (UP) application in the EUFM formula F with a new term (Boolean) variable and then adds *external constraints for functional consistency*. For example, the UF application $g(a_1, b_1)$ will be replaced by a new term variable c_1, another application of the same UF, $g(a_2, b_2)$, will be replaced by a new term variable c_2. Then, the resulting EUFM formula F' will be extended as $[(a_1 = a_2) \wedge (b_1 = b_2) \Rightarrow (c_1 = c_2)] \Rightarrow F'$. Note that the new formula is equivalent to $(a_1 = a_2) \wedge (b_1 = b_2) \wedge \neg(c_1 = c_2) \vee F'$, so that the new term variables, c_1 and c_2, appear in a negated equation. In the nested-*ITE*s scheme, the first application of a UF will still be replaced by a new term variable c_1. However, the second will be replaced by $ITE((a_2 = a_1) \wedge (b_2 = b_1), c_1, c_2)$, where c_2 is a new term variable. A third application, $g(a_3, b_3)$, will be replaced by $ITE((a_3 = a_1) \wedge (b_3 = b_1), c_1, ITE((a_3 = a_2) \wedge (b_3 = b_2), c_2, c_3))$, where c_3 is a new term variable, and so on. UPs are eliminated similarly, but using new Boolean variables. In the general case of each scheme, the formulas that express equality of arguments of UF (UP) applications with k arguments, will be conjunctions of k equations, one for each pair of corresponding arguments. To avoid creating circular dependencies *when using the nested-ITE scheme, UFs and UPs have to be eliminated based on their topological order*, i.e., all applications of a given UF (UP) have to be eliminated from the arguments of another application of the same UF (UP), before that application is eliminated. Otherwise, the equations between corresponding arguments will lead to cyclic dependency.

We can prove liveness by a modified version of the safety correctness criterion—by symbolically simulating the implementation for a finite number of steps, n, and proving that:

$$equality_1 \vee equality_2 \vee \ldots \vee equality_{n \times k} = true$$

where k is the issue width of the implementation. The formula proves that n steps of the implementation match between 1 and $n \times k$ steps of the specification, when the implementation starts from an arbitrary initial state that may be restricted by invariant constraints. Note that (1) *guarantees that the implementation has made at least one step*, while the safety correctness criterion allows the implementation to stay in its initial state when formula $equality_0$ (checking whether the implementation matches the initial state of the specification) is *true*. The correctness formula is generated automatically in the same way as the formula for safety, except that the implementation and the specification are symbolically simulated for many steps, and formula $equality_0$ is not included. The minimum number of steps, n, to symbolically simulate

the implementation, can be determined experimentally, by trial and error, or can be provided by the user, based on knowledge about the stalling and squashing behavior of the implementation. We can also prove the liveness criterion (1) indirectly [40, 45]—by first proving safety, thus inductively the implementation correctness for n steps, and then using Positive Equality to prove that $equality_0$ will be *false* after n steps; this indirect method resulted in 4 orders of magnitude speedup when proving liveness of complex processors.

2.2 The RazorII Mechanism for Soft-Error Tolerance

The RazorII mechanism [12] is based on the flip-flop (FF) design shown in Fig. 2. It uses a single positive level-sensitive latch, augmented with a transition detector (TDetector) controlled by a detection clock (DC), whose rising edge is delayed relative to the rising edge of the regular clock, CLK. Timing errors are detected by monitoring the internal latch node for spurious transitions. A legitimate transition occurs when data is setup to the latch input before the rising edge of the clock. In this case, the output Q of the latch transitions at the rising edge after a delay equal to the clock-to-Q (CLK-Q) delay of the latch, to reflect the state of data being captured. In order to prevent legitimate transitions being flagged as timing errors, a short negative pulse on the detection clock is used to disable the transition detector for at least the duration of the CLK-Q delay after the rising edge, as shown in the figure. However, if the input data transitions after the rising clock edge, during transparency, the transition of latch node, N, occurs when the transition detector is enabled and results in assertion of signal Error. That signal triggers the architectural replay mechanism to restore correct state within the pipeline, as illustrated in Fig. 3. The design of the RazorII FF naturally allows it to detect Single Event Upsets (SEUs) within the flip-flop and in the combinational logic.

The authors of [12] implemented the RazorII mechanism in a 64-bit, 7-stage pipelined Alpha processor in order to detect timing errors and SEUs—see Fig. 3. The pipeline has two instruction fetch stages (IF1, IF2), instruction decode stage (ID), an integer execution unit (EX), memory access stage (MEM), and write-back stage (WB). All pipeline registers have RazorII protection against state corruption due to SEUs. The error pins of all the RazorII FFs in each pipeline stage are ORed together, and the result is propagated and ORed with that of the next stage. This allows the composite error signal for the entire pipeline to be evaluated on a per-stage basis.

In the pipelined processor in Fig. 3, the read and write paths to the instruction and data cache (ICACHE and DCACHE), the register file, and the program status register are not time-critical, and so do not require RazorII protection. Error Correcting Codes (ECCs) are used to recover from SEUs in the caches and the register file. The program status registers are protected using Triple Module Redundancy (TMR) [29]. The key concept of TMR is to use three blocks of logic for the same computation. Then, a majority voting circuit selects as output the data value produced by at least two of the three blocks. TMR allows SEU errors in the architectural state registers to be corrected on-the-fly, without an extra recovery mechanism.

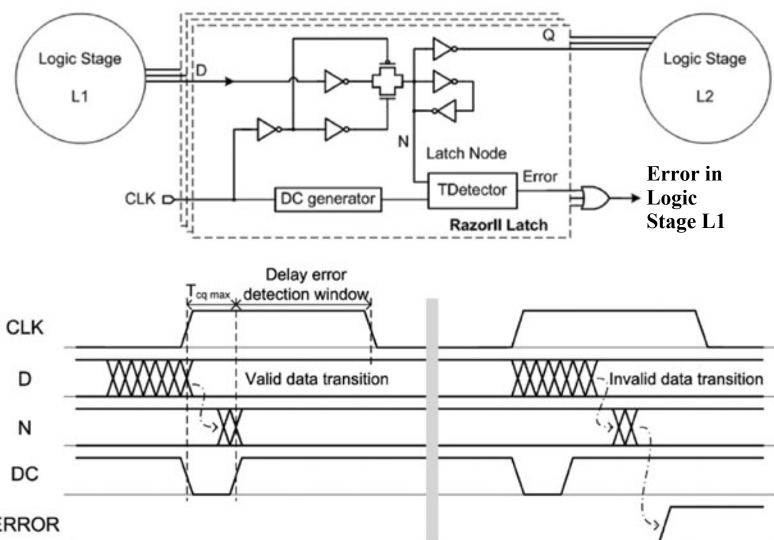


Fig. 2. The RazorII fault-detecting flip-flop, and a conceptual timing diagram for its operation. (This figure is a modified version of one from [12].)

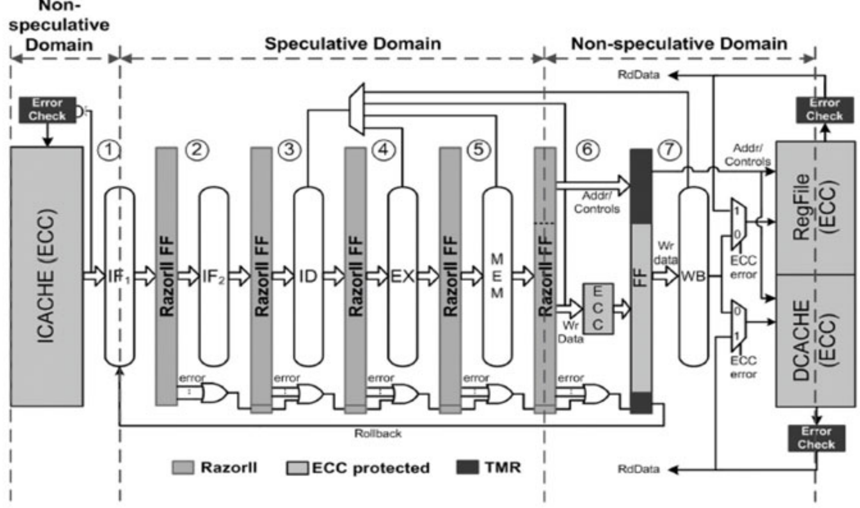


Fig. 3. Pipelined processor with the RazorII mechanism. Since the Error signals from the RazorII FFs in each pipeline stage are OR-ed together, this scheme will detect any number of SEUs in a pipeline stage. (This figure is from [12].)

Replay is achieved by check-pointing the Program Counter (PC) register in the WB stage of the pipeline. The PC is passed along the processor pipeline together with its corresponding instruction. When an instruction is in the WB stage and a soft error has occurred during the execution of that instruction, the entire pipeline is flushed and the PC in the fetch stage is overwritten with the PC in the WB stage. Normal instruction execution resumes from then on. The PC in the WB stage is protected from SEUs through TMR. Since an erroneous instruction is re-executed through the pipeline during replay, the same instruction can suffer repeated timing errors. Hence, it is necessary to monitor the instruction being replayed to avoid a deadlock. When the number of replay iterations for the same instruction reaches a certain threshold, called the *replay limit*, the clock frequency is halved for as many cycles as to allow that instruction to complete its execution. Thus, for a replay limit of 1, every timing error is accompanied by recovery at half the clock frequency. For a replay limit of n, an errant instruction is replayed $n - 1$ times at the same frequency, if required, before the frequency is halved in the nth iteration. The majority of timing errors at the PoFF are actually caused due to transient events, such as cross-coupling noise, and so disappear during replay [12]. Hence, it is expected that for most timing errors, replaying the erroneous instruction just once will be sufficient for completion, without having to reduce the clock frequency.

The authors of [12] observed from their silicon measurement results that 60% of failing instructions were executed to completion in the first replay iteration without reducing the clock frequency. They used an externally programmable replay limit. Furthermore, their silicon measurements showed that when the supply voltage is lowered to the PoFF, the frequency of failing instructions was approximately 1 in 100 million completed instructions, thus resulting in an insignificant overhead of the replay [5]. Those authors tested a fabricated copy of their pipelined processor with the RazorII mechanism in a radioactive environment. RazorII was able to detect and correct all faults, allowing the processor to compute correct results for a test sequence with tens of millions of instructions.

3 Formal Verification of Pipelined Processors with Mechanisms for Soft-Error Tolerance

When implementing at a high level of abstraction and formally verifying pipelined processors with mechanisms for soft-error tolerance, our requirements were to be able to: 1) formally verify both safety and liveness; 2) formally verify processors with different pipeline structure, including single-issue pipelined, VLIW, and superscalar designs where in the latter an instruction can have many execution paths; 3) easily apply these techniques at a high level of abstraction, in order to abstract any mechanism for detection of soft errors; and 4) exploit the property of Positive Equality in order to classify as many term variables as p-terms, hence allowing our automatic decision procedure to treat them as distinct constants and thus achieve orders of magnitude speedup.

To satisfy all these requirements, we abstracted the logic for detecting soft errors in the execution of each instruction in every pipeline stage. We did that with a generator of arbitrary values [33] that produces a fresh Boolean variable during every clock

cycle to indicate whether a soft error has occurred during that clock cycle. We introduced a different generator of arbitrary values (that will produce a unique sequence of fresh Boolean variables) for each instruction in every pipeline stage. That is, processors that can have several instructions in execution in each pipeline stage, such as superscalar and VLIW processors, will have as many such generators of arbitrary values in each pipeline stage as the number of instructions that can be in simultaneous execution in that stage. This will abstract both possible outcomes of occurrence and non-occurrence, respectively, of a soft error during the execution of each symbolic instruction in each pipeline stage for each iteration (i.e., original execution or replay) of the instruction, hence representing all possible execution scenarios during the symbolic simulation of the implementation processor.

We also need to enforce the constraint that when the replay limit is reached for an instruction, then the instruction will be executed without soft errors in the next iteration. This is based on the assumption [12] that in the actual implementation of the processor, the clock frequency will be halved when the replay limit is reached, and that guarantees that the logic in every pipeline stage will become resistant to soft errors, which will ensure the completion of the instruction in the next iteration. To model this behavior, we introduce an additional set of bit-level signals that propagate with each instruction in the high-level model of the implementation processor, and represent a one-hot encoding (where only one of the bit-level signals is 1 and the rest are 0s) that indicates the exact iteration of an instruction up to the given replay limit plus one. A constraint is imposed:

(*c1*) for each instruction in each pipeline stage that if the replay limit has been exceeded then there will be no soft errors, i.e., the output value of the generator of arbitrary values that abstracts the occurrence of a soft error for that instruction in a particular pipeline stage in a given clock cycle will be *false*.

This constraint is imposed for every clock cycle of symbolic simulation of the implementation processor—during both regular symbolic simulation and flushing, and for the proof of both safety and liveness.

We also imposed two invariant constraints for every instruction in every pipeline stage:

(*i1*) that the above bit-level signals that represent the iteration number form a one-hot pattern—this was done by using a condition that the disjunction of all legitimate one-hot patterns of these bit-level signals is *true*; and

(*i2*) that if the replay limit has been exceeded then no soft errors have occurred during the execution of this instruction in earlier pipeline stages, i.e., the bit-level signals that propagate with the instruction and indicate whether soft errors have occurred earlier are all *false*.

To check an invariant constraint, we impose it for the initial symbolic state of the implementation processor, then simulate the processor symbolically for one clock cycle, and verify that the same condition is satisfied. If so, then these constraints are imposed for the initial implementation state when proving both safety and liveness.

The outcomes of whether a soft error has occurred in any stage of executing an instruction are OR-ed together to form the combined outcome of a soft-error occurrence during the execution of that instruction. The resulting signal is used in the last

pipeline stage to squash subsequent instructions, and to control the replay of the instruction that encountered a soft error. Namely, the condition that a soft error has occurred is used in the first pipeline stage to control multiplexors that will select for re-execution the instruction from the last pipeline stage. This is done by shifting the one-hot bit pattern (that indicates the exact iteration of an instruction) by one bit and filling the least significant bit with a 0. In the event that a soft error has not occurred, then executed is the newly fetched instruction, such that the corresponding bit-level pattern with a one-hot encoding is set to have a 1 in its least significant position and 0s elsewhere.

If an instruction is stalled in a given pipeline stage during a particular clock cycle, then the signal indicating whether a soft error has occurred for that instruction in that stage in that clock cycle is set to 0. That signal is similarly set to 0 if the instruction is squashed, e.g., when a preceding instruction has a branch misprediction [33], a data-value misprediction [47], a raised exception [33], or a soft error has occurred during the execution of the instruction that is currently being completed in the last pipeline stage.

Instead of a one-hot encoding, we can use the actual bit-level pattern of the binary representation of the number of iterations. Then, we no longer need the first invariant constraint (*i1*) for a legitimate one-hot pattern, but still need the constraint (*c1*) and the second invariant constraint (*i2*). Also, if a soft error has occurred and so the instruction is replayed, then we need to use an incremeter (an adder that increments a binary value with 1) to form the next bit-level pattern for the execution iteration.

The use of a one-hot bit-level representation of the number of iterations simplifies the resulting formulas for the invariants, the constraints, and the correctness conditions for safety and liveness. We call this high-level *design for formal verification*. (This concept is analogous to that of design for testability in the testing community.) Note that the one-hot representation of the number of iterations can be replaced with any representation of that number in the actual synthesized processor, as long as there is a one-to-one correspondence between patterns and it is guaranteed that when the replay limit is exceeded then the corresponding number of iterations is used to trigger a mechanism that guarantees that there will be no soft errors in the next iteration for that instruction, e.g., by reducing the clock frequency by half [12].

The above techniques are applied only to the high-level model of the pipelined/superscalar/VLIW implementation processor. The non-pipelined specification processor that defines the correct instruction semantics is not modified. The above abstractions are applicable to all processor architectures: single-issue pipelined, superscalar, or VLIW.

Lemma 1. The presented abstraction of the mechanism for soft-error tolerance is sound and complete.

Sketch of the proof: The presented abstraction of the logic for detection of soft errors represents the most general symbolic execution scenario—a soft error could occur for every instruction in every stage in every clock cycle unless the replay limit is exceeded—and thus accounts for all possible execution scenarios that satisfy the assumption that after the replay limit is exceeded then there will be no soft errors in the next clock cycle, and so the instruction will be definitely completed then. □

We abstract multicycle functional units with place holders [33], which are described in AbsHDL, our high-level hardware description language. The place holders are

constructed so that they exhibit enough of the timing characteristics of the original functional units, such that the correctness of the abstract processor with place holders will imply the correctness of the actual implementation with the original functional units. For example, if a functional unit has a fixed latency of n cycles, we can abstract it with a model that has a chain of $n - 1$ latches situated in the stage of the original functional unit, as shown in Fig. 4.

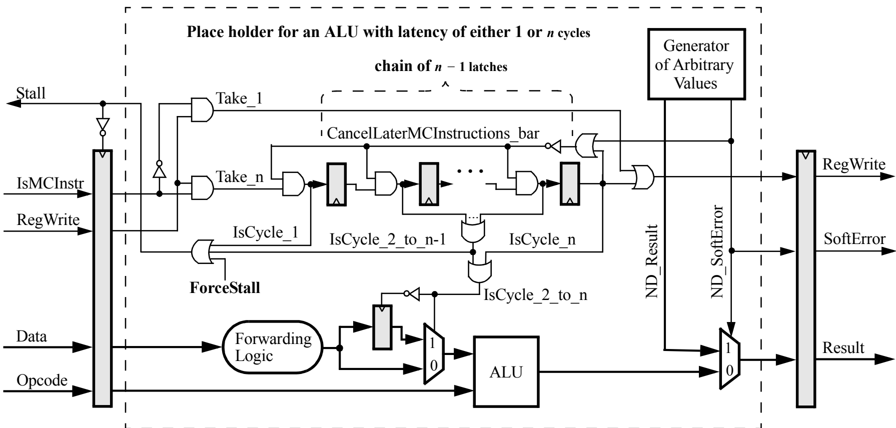


Fig. 4. Abstraction for a functional unit that can take either 1 or n cycles to complete a computation. ALU is the uninterpreted function that abstracts the semantics of the computation performed by the functional unit, and is replaced by a memory model when a multicycle memory is abstracted.

Signal ForceStall in Fig. 4 is used to implement Burch's controlled flushing [10], and helps eliminate the ambiguity in the instruction flow during flushing, thus significantly simplifying the EUFM correctness formula. We set that signal to *false* during the one cycle of regular symbolic simulation of the processor when checking safety. However, we set signal ForceStall to *true* during flushing in order to force the original signal Stall to *true*, thus stalling the instructions in the previous pipeline stages; this continues until the original signal Stall is guaranteed to evaluate to *false*. Then, we repeat the sequence: we set ForceStall to *false* for 1 clock cycle in order to allow the instructions in the previous stages to advance, and so let a new instruction in the stage of the multicycle functional unit; then we set signal ForceStall to *true* until the original signal Stall is again guaranteed to evaluate to *false*, and so on.

In Fig. 4, the latency is 1 cycle when signal Take_1 is *true* (i.e., RegWrite is *true* and IsMCInstr is *false*) or n cycles when signal Take_n is *true* (i.e., RegWrite is *true* and IsMCInstr is *true*). The chain of $n - 1$ latches is used to delay signal IsMCInstr for n cycles when propagating through the stage of the original functional unit. Signal Stall is used to stall the previous pipeline stages when a multicycle computation is in cycles 1 through $n - 1$ of its execution. By introducing the feedback loop created by signal CancelLaterMCInstructions, we avoid the need to impose and check the invariant that at most one latch in the chain can have value *true*. Signal ForceStall is used to implement Burch's controlled flushing.

To abstract the detection of soft errors, we introduce a generator of arbitrary values that produces an arbitrary Boolean variable ND_SoftError to abstract the occurrence of a soft error in a clock cycle, and an arbitrary term ND_Result to abstract the result of the multicycle computation if a soft error occurs—in that case any subsequent symbolic instructions in execution in the same functional unit are cancelled.

4 Results

The experiments were conducted on a workstation with a 3.3-GHz six-core Intel Xeon processor, and 24 GB of memory, running Red Hat Enterprise Linux v5.5. (Only a single core was used for each experiment.) The symbolic simulation was done with the term-level symbolic simulator TLSim [46]. Translation of the EUFM correctness formula to an equivalent Boolean formula was done with the decision procedure EVC [46]. The SAT solver rsat v3.1 [20, 21] was used for solving the CNF formulas.

Table 1. Experimental results. MSET stands for Mechanism for Soft Error Tolerance, while the original design does not have such a mechanism

Benchmark	Version	CPU Time [s] to Check Safety	CPU Time [s] to Check Liveness
`1dlx_c`	Original	0.06	0.52
	MSET	0.07	0.65
`2dlx_ca`	Original	0.18	2.15
	MSET	0.24	2.72
`2dlx_cc_mc_ex_bp`	Original	0.90	6.2
	MSET	1.02	7.3
`9vliw_bp_mc_ex_9stages_iq5`	Original	612	3,243
	MSET	723	4,495

The experiments were to formally verify safety and liveness of the benchmarks: `1dlx_c`, a single-issue 5-stage pipelined DLX [15], modeled as described in [32]; `2dlx_ca`, a dual-issue superscalar DLX, with one complete and one ALU pipeline [32]; `2dlx_cc_mc_ex_bp`, a dual-issue superscalar DLX, with 2 complete pipelines, exceptions, branch prediction, and multicycle functional units [33]; and `9vliw_bp_mc_ex_9stages_iq5`, a 9-stage, 9-wide VLIW processor that imitates the Intel Itanium [16, 27] in features such as predicated execution, register remapping, advanced loads, branch prediction, multicycle functional units, exceptions, and a 5-entry instruction queue (a simpler version of this processor with fewer pipeline stages and no instruction queue was formally verified in [34, 36]). The abstraction function was computed by controlled flushing [10], where the user provides a stalling schedule to override the processor stall signals, thus eliminating the ambiguity of the instruction flow during flushing, and producing a simpler EUFM correctness formula.

Table 1 presents the results. In the two benchmarks with multicycle functional units, the Instruction Memory, the ALUs in the Execution stage, and the Data Memory could take either 1 or 4 clock cycles to fetch and compute a result, respectively. Liveness was formally verified with the indirect method from [40]. As can be seen from the table, modeling the abstracted mechanism for soft error tolerance, including a replay mechanism and abstraction of the logic that detects soft errors, resulted in negligible overhead when proving safety—up to about 15% at the most. The overhead was up to about one third when formally verifying liveness, and was still acceptable.

5 Related Work

The only related work that we know of is by Alizadeh et al. [4]. They use an uninterpreted predicate with no arguments, i.e., a Boolean variable, to abstract the occurrence of a timing error in a pipeline stage of a DLX processor. According to that paper, they introduce one such uninterpreted predicate per pipeline stage, which will correctly capture the non-deterministic occurrence of a timing error in the first cycle of symbolic simulation, but then the same variable will be reused for this purpose for all subsequent instructions that pass through that stage, which will be incorrect. They formally verify only safety for a 5-stage DLX comparable to our `1dlx_c`, while we presented techniques for formally verifying both safety and liveness, including of designs with multicycle functional units. Also, we showed results for complex pipelined, superscalar, and VLIW processors with exceptions, branch prediction, and multicycle functional units.

6 Conclusion

We presented techniques for design at a high level of abstraction and formal verification of pipelined/superscalar/VLIW processors with built-in mechanisms for soft-error and timing-error tolerance. Our requirements were to be able to: 1) formally verify both safety and liveness; 2) formally verify processors with different pipeline structure, including single-issue pipelined, VLIW, and superscalar designs where in the latter an instruction can have many execution paths; 3) ease of modeling at a high level of abstraction, applicability to different mechanisms for detection of soft errors, and simplicity of the formal verification; and 4) exploit the property of Positive Equality in order to classify as many term variables as p-terms, hence allowing our automatic decision procedure to treat them as distinct constants and thus achieve orders of magnitude speedup. To satisfy all these requirements, we abstracted the logic for detecting of soft errors in the execution of an instruction after each pipeline stage with a generator of arbitrary values. An additional set of bit-level signals representing a one-hot encoding was used to indicate the exact iteration of an instruction up to a given replay limit plus one, with a constraint imposed that when the replay limit is exceeded then there will be no soft errors, based on the assumption that in the actual implementation the clock frequency is halved when the replay limit is exceeded and that guarantees a completion of an instruction with no soft errors [12]. We applied the techniques to formally verify both safety and liveness of single-issue pipelined and dual-issue superscalar processors, as well as VLIW processors that imitate the Intel Itanium in many features.

References

[1] Aagaard, M.D., Day, N.A., Lou, M.: Relating multi-step and single-step microprocessor correctness statements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517. Springer, Heidelberg (2002)

[2] Aagaard, M.D., Cook, B., Day, N.A., Jones, R.B.: A framework for superscalar microprocessor correctness statements. Software Tools for Technology Transfer (STTT) 4(3) (May 2003)

[3] Ackermann, W.: Solvable Cases of the Decision Problem. North-Holland, Amsterdam (1954)

[4] Alizadeh, B., Gharehbaghi, A.M., Fujita, M.: Pipelined Microprocessors Optimization and Debugging. In: Sirisuk, P., Morgan, F., El-Ghazawi, T., Amano, H. (eds.) Reconfigurable Computing: Architectures, Tools and Applications. LNCS, vol. 5992, pp. 435–444. Springer, Heidelberg (2010)

[5] Blaauw, D., Das, S.: CPU, Heal Thyself: A Fault-Monitoring Microprocessor Design Can Save Power or Allow Overclocking. IEEE Spectrum (August 2009), http://spectrum.ieee.org/semiconductors/processors/cpu-heal-thyself/0

[6] Bouajila, A., Zeppenfeld, J., Stechele, W., Herkersdorf, A., Bernauer, A., Bringmann, O., Rosenstiel, W.: Organic Computing at the System on Chip Level. In: IFIP International Conference on Very Large Scale Integration (VLSI-SoC 2006), pp. 338–341 (2006)

[7] Bryant, R.E., German, S., Velev, M.N.: Processor Verification Using Efficient Reductions of the Logic of Uninterpreted Functions to Propositional Logic. ACM Transactions on Computational Logic 2(1) (2001)

[8] Bryant, R.E., Velev, M.N.: Boolean Satisfiability with Transitivity Constraints. ACM Transactions on Computational Logic (TOCL) 3(4), 604–627 (2002)

[9] Burch, J.R., Dill, D.L.: Automated Verification of Pipelined Microprocessor Control. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818. Springer, Heidelberg (1994)

[10] Burch, J.R.: Techniques for Verifying Superscalar Microprocessors. In: Design Automation Conference (June 1996)

[11] Burcin, A.: RAD750, BAE Systems (December 2002), http://www.aero.org/conferences/mrqw/2002-papers/A_Burcin.pdf

[12] Das, S., Tokunaga, C., Pant, S., Ma, W.-H., Kalaiselvan, S., Lai, K., Bull, D.M., Blaauw, D.T.: RazorII: In Situ Error Detection and Correction for PVT and SER Tolerance. IEEE Journal of Solid-State Circuits 44(1), 32–48 (2009)

[13] Goel, A., Sajid, K., Zhou, H., Aziz, A., Singhal, V.: BDD Based Procedures for a Theory of Equality with Uninterpreted Functions. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427. Springer, Heidelberg (1998)

[14] Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust Sat-Solver. In: Design, Automation, and Test in Europe (DATE 2002), pp. 142–149 (March 2002)

[15] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach, 3rd edn. Morgan Kaufmann Publishers, San Francisco (2002)

[16] Intel Corporation: IA-64 Application Developer's Architecture Guide (May 1999), http://developer.intel.com/design/ia-64/architecture.htm

[17] Lahiri, S., Pixley, C., Albin, K.: Experience with Term Level Modeling and Verification of the M·CORE™ Microprocessor Core. In: International Workshop on High Level Design, Validation and Test (HLDVT 2001) (2001)

[18] Le Berre, D., Simon, L.: Results from the SAT 2004 SAT Solver Competition. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 321–344. Springer, Heidelberg (2005)
[19] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: 38th Design Automation Conference (DAC 2001) (June 2001)
[20] Pipatsrisawat, K., Darwiche, A.: A Lightweight Component Caching Scheme for Satisfiability Solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
[21] Pipatsrisawat, K., Darwiche, A.: A New Clause Learning Scheme for Efficient Unsatisfiability Proofs. In: Twenty-Third AAAI Conference on Artificial Intelligence, pp. 1481–1484 (July 2008)
[22] Pnueli, A., Rodeh, Y., Strichman, O., Siegel, M.: The Small Model Property: How Small Can It Be? Journal of Information and Computation 178(1) (2002)
[23] Rotenberg, E.: AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In: Annual International Symposium on Fault-Tolerant Computing (June 1999)
[24] Ryan, L.: Siege SAT Solver v.4, http://www.cs.sfu.ca/~loryan/personal/
[25] Su, Y.-S., Chang, P.-H., Chang, S.-C., Hwang, T.: Synthesis of a Novel Timing-Error Detection Architecture. Transactions on Design Automation of Electronic Systems (TODAES) 13(1) (January 2008)
[26] Subramanian, V., Bezdek, M., Avirneni, N.D., Somani, A.: Superscalar Processor Performance Enhancement Through Reliable Dynamic Clock Frequency Tuning. In: Annual IEEE/IFIP International Conference on Dependable Systems and Networks (2007)
[27] Sharangpani, H., Arora, K.: Itanium processor microarchitecture. IEEE Micro 20(5), 24–43 (2000)
[28] Tschanz, J., Kim, N.S., Dighe, S., Howard, J., Ruhl, G., Vanga, S., Narendra, S., Hoskote, Y., Wilson, H., Lam, C., Shuman, M., Tokunaga, C., Somasekhar, D., Tang, S., Finan, D., Karnik, T., Borkar, N., Kurd, N., De, V.: Adaptive Frequency and Biasing Techniques for Tolerance to Dynamic Temperature-Voltage Variations and Aging. In: IEEE International Solid-State Circuits Conference (ISSCC 2007), pp. 292–604 (February 2007)
[29] Van Gils, W.J.: A Triple Modular Redundancy Technique Providing Multiple-Bit Error Protection Without Using Extra Redundancy. IEEE Trans. Computers C-35(7), 623–631 (1986)
[30] Velev, M.N., Bryant, R.E.: Bit-Level Abstraction in the Verification of Pipelined Microprocessors by Correspondence Checking. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 18–35. Springer, Heidelberg (1998)
[31] Velev, M.N., Bryant, R.E.: Exploiting Positive Equality and Partial Non-Consistency in the Formal Verification of Pipelined Microprocessors. In: 36th Design Automation Conference (DAC 1999), pp. 397–401 (June 1999)
[32] Velev, M.N., Bryant, R.E.: Superscalar Processor Verification Using Efficient Reductions of the Logic of Equality with Uninterpreted Functions to Propositional Logic. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 37–53. Springer, Heidelberg (1999)
[33] Velev, M.N., Bryant, R.E.: Formal Verification of Superscalar Microprocessors with Multicycle Functional Units, Exceptions, and Branch Prediction. In: 37*th* Design Automation Conference (DAC 2000), pp. 112–117 (June 2000)

[34] Velev, M.N.: Formal verification of VLIW microprocessors with speculative execution. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 296–311. Springer, Heidelberg (2000)
[35] Velev, M.N.: Automatic Abstraction of Memories in the Formal Verification of Superscalar Microprocessors. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 252–267. Springer, Heidelberg (2001)
[36] Velev, M.N., Bryant, R.E.: Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. Journal of Symbolic Computation (JSC) 35(2), 73–106 (2003)
[37] Velev, M.N.: Automatic Abstraction of Equations in a Logic of Equality. In: Cialdea Mayer, M., Pirri, F. (eds.) TABLEAUX 2003. LNCS(LNAI), vol. 2796, pp. 196–213. Springer, Heidelberg (2003)
[38] Velev, M.N.: Using Automatic Case Splits and Efficient CNF Translation to Guide a SAT-Solver When Formally Verifying Out-of-Order Processors. In: Artificial Intelligence and Mathematics (AI&MATH 2004), pp. 242–254 (January 2004)
[39] Velev, M.N.: Efficient Translation of Boolean Formulas to CNF in Formal Verification of Microprocessors. In: Asia and South Pacific Design Automation Conference (ASP-DAC 2004), pp. 310–315 (January 2004)
[40] Velev, M.N.: Using Positive Equality to Prove Liveness for Pipelined Microprocessors. In: Asia and South Pacific Design Automation Conference (ASP-DAC 2004) (January 2004)
[41] Velev, M.N.: Exploiting Signal Unobservability for Efficient Translation to CNF in Formal Verification of Microprocessors. In: Design, Automation and Test in Europe (2004)
[42] Velev, M.N.: Encoding Global Unobservability for Efficient Translation to SAT. In: International Conference on Theory and Applications of Satisfiability Testing (May 2004)
[43] Velev, M.N.: Comparative Study of Strategies for Formal Verification of High-Level Processors. In: 22nd International Conference on Computer Design (ICCD 2004) (October 2004)
[44] Velev, M.N.: Comparison of Schemes for Encoding Unobservability in Translation to SAT. In: Asia & South Pacific Design Automation Conference (ASP-DAC 2005) (January 2005)
[45] Velev, M.N.: Automatic Formal Verification of Liveness for Pipelined Processors with Multicycle Functional Units. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 97–113. Springer, Heidelberg (2005)
[46] Velev, M.N., Bryant, R.E.: TLSim and EVC: A Term-Level Symbolic Simulator and an Efficient Decision Procedure for the Logic of Equality with Uninterpreted Functions and Memories. International Journal of Embedded Systems 1(1/2) (2005)
[47] Velev, M.N.: Using Abstraction for Efficient Formal Verification of Pipelined Processors with Value Prediction. In: 7th International Symposium on Quality Electronic Design (ISQED 2006), pp. 51–56 (March 2006)
[48] Velev, M.N., Gao, P.: Exploiting Hierarchical Encodings of Equality to Design Independent Strategies in Parallel SMT Decision Procedures for a Logic of Equality. In: IEEE High Level Design Validation and Test Workshop (HLDVT 2009) (November 2009)
[49] Velev, M.N., Gao, P.: A Method for Debugging of Pipelined Processors in Formal Verification by Correspondence Checking. In: 15th Asia and South Pacific Design Automation Conference (ASP-DAC 2010), pp. 619–624 (January 2010)

Formal Verification of Tokeneer Behaviours Modelled in fUML Using CSP

Islam Abdelhalim, James Sharp, Steve Schneider, and Helen Treharne

Department of Computing, University of Surrey

Abstract. Much research work has been done on formalizing UML diagrams, but less has focused on using this formalization to analyze the dynamic behaviours between formalized components. In this paper we propose using a subset of fUML (Foundational Subset for Executable UML) as a semi-formal language, and formalizing it to the process algebraic specification language CSP, to make use of FDR as a model checker. Our formalization includes modelling the asynchronous communication framework used within fUML. This allows different interpretations of the communications model to be evaluated. To illustrate the approach, we use the modelling of the Tokeneer ID Station specifications into fUML, and formalize them in CSP to check if the model is deadlock free.

1 Introduction

The OMG (Object Management Group) has developed the fUML (Foundational Subset for Executable UML) [1] specification with the purpose of enabling compliant models to be transformed into various executable forms for verification, integration, and deployment. The specification of the execution model incorporates a degree of genericity. This is achieved mainly by defining explicit semantic variation points. A particular execution tool can then realize specific semantics by providing specifications for any of those points. The *semantics of inter-object communications mechanisms* is one of the semantic variation points. The choice of the implementation of such a point may affect the execution model of the system. We use formal model checking to evaluate an implementation of the inter-object communications mechanism and its compatibility with an fUML model. This, in turn, can ensure that faults within an fUML model are detected during the early stages of software development lifecycle, which provide significant savings in cost compared with rectifying errors after the system has been implemented.

We formalize fUML models into the process algebraic specification language CSP [2] with the purpose of checking deadlocks in the models, which could happen if all objects are waiting to accept signals from each other. To check such a property, we modelled the inter-object communications mechanism into CSP. Then we used FDR (Failures-Divergences Refinement) [3] to handle the model checking and report the deadlock scenarios (if found). Potentially, we

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 371–387, 2010.
© Springer-Verlag Berlin Heidelberg 2010

can make use of this methodology to allow software engineers, who do not have specialist mathematical knowledge, to formally check their semi-formal (fUML) models.

Much research exists on translating a UML model to a formal model (see Section 8) but many approaches have imposed restrictions on the UML diagrams and notation used. Our previous work on transforming xUML (Executable UML) [4] to $CSP \parallel B$ [5] benefits from the fact that using xUML as a starting point means that the restrictions are those imposed by the language itself. xUML allows models to be executed and tools supporting xUML have meta-models [6]. In this paper we propose to examine the translation between fUML and CSP. fUML benefits from a standardized meta-model. Moreover, the complexity of the queuing mechanism in xUML (each object has a separate signal queue for all the objects in the system including itself which has the highest priority) leads to more complex CSP models, and thus more heavy processes to be analyzed by FDR. Also the syntax of the fUML activity diagrams allowed for modelling the internal choices, which was not possible using xUML. The fUML models yield more abstract formal models than were possible from xUML translation.

In order to validate our approach, we modelled part of the Tokeneer ID Station project [7] into fUML. We then developed a group of mapping rules (maps between the fUML activity diagrams elements and CSP) to manually translate the fUML into CSP, and checked whether the CSP model is deadlock free. If the CSP is deadlock free it gives us confidence that the corresponding fUML model is also deadlock free. In this paper we are focusing on a subset of the fUML activity diagram notation, and this has been guided by the Tokeneer case study. Formalizing fUML class diagrams is beyond the scope of this paper, as we want to check the dynamic behaviour of the system. Also the runtime objects creation and deletion are not covered in this work.

Overall the main novelty of this work is the proposed approach to model the asynchronous communication between objects. Also to our knowledge, this is the first attempt to analyze fUML models (by formalizing them into CSP) and addressing the semantic variation points issue in the fUML standard.

We assume the reader of this paper has good background knowledge of the UML 2 standard, CSP, and FDR.

The rest of this paper is organised as follows. In Section 2, we give a background to the fUML standard and the CSP syntax used in this paper. In Section 3, we introduce the Tokeneer case study briefly and include part of the used fUML diagrams. In Section 4, we describe the mapping rules we developed to map between the fUML activity diagram elements and CSP. In Section 5, we describe how we model the asynchronous communication between the fUML active objects in CSP. In Section 6, we describe the CSP model for a selected part of Tokeneer fUML activity diagram. In Section 7, we discuss the deadlock checking for the formalized fUML model and the generated results from FDR. Finally, we discuss related work and conclude in Sections 8 and 9 respectively.

2 Background

2.1 fUML

fUML (Foundational Subset for Executable UML) [1] is an OMG standard acting as an intermediary between "surface subsets" of UML models and platform executable languages. The OMG defines the fUML subset by specifying the modifications to the original abstract syntax (of UML 2) of the class and activity diagrams, which allows fUML models to be transformed to various executable forms. These modifications are specified in clause 7 of the standard by merging/excluding some packages in the UML 2 specification [8], as well as adding new constraints. As defined in the fUML standard, we are listing below some modifications to UML 2 that are relevant to our Tokeneer fUML model; all of them are related to the fUML activity diagrams since our goal is to capture the behaviours of our model:

- Central buffer nodes are excluded from fUML because they were judged to be unnecessary for the computational completeness of fUML.
- Variables are excluded from fUML because the passing of data between actions can be achieved using object flows.
- Exception handlers are not included in fUML because exceptions are not included in fUML.
- Opaque actions are excluded from fUML since, being opaque, they cannot be executed.
- Value pins are excluded from fUML because they are redundant with using value specifications to specify values.

The operational semantics of fUML is an executable model with methods written in Java, with a mapping to UML activity diagrams. The declarative semantics of fUML is specified in first order logic and based on PSL (Process Specification Language).

Inter-object Communication Mechanism in fUML
This part gives an overview of the semantics of the inter-object communication in fUML as defined by clause 8 in the standard [1]. Such communication is conducted between active objects only. Active objects in fUML communicate asynchronously via signals (kind of classifier). This is achieved by associating an object activation with each object which handles the dispatching of asynchronous communications received by its active object. Figure 1 shows the structure related to the object activation. The object activation maintains two main lists: the first list (event pool) holds the incoming signal instances waiting to be dispatched, and the second list (waiting event accepters) holds the event accepters that have been registered by the executing classifier behaviour. Event accepters are allowable signals with respect to the current state of the active object. The fUML standard permits the specifier (tool implementer) to define a suitable dispatching mechanism for signals within the *event pool* (semantic variation point). The default dispatching behaviour, as described in [1], dispatches events on a FIFO (first-in first-out) basis.

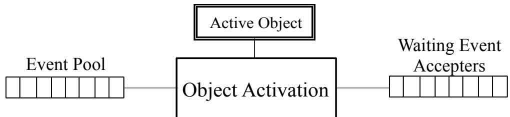


Fig. 1. Object Activation Structure

2.2 CSP

CSP (Communication Sequential Processes) [2] is a modelling language that allows the description of systems of interacting processes using a few language primitives. Processes execute and interact by means of performing events drawn from a universal set Σ. Some events are of the form $c.v$, where c represents a channel and v represents a value being passed along that channel. We considered the following subset of the CSP syntax:

$$P ::= a \rightarrow P | c?x \rightarrow P(x) | d!v \rightarrow P | P_1 \Box P_2 | P_1 \sqcap P_2$$
$$| P_1 \underset{A\ B}{\|} P_2 | P \setminus A | P_1;\ P_2 | \text{ if } b \text{ then } P_1 \text{ else } P_2$$

The CSP process $a \rightarrow P$ initially allows event a to occur and then behave subsequently as P. The input process $c?x \rightarrow P(x)$ will accept a value x along channel c and then behave subsequently as $P(x)$. The output process $c!v \rightarrow P$ will output v along channel c and then behave as P. Channels can have any number of message fields, combination of input and output values.

The choice $P_1 \Box P_2$ offers an external choice between processes P_1 and P_2 whereby the choice is made by the environment. Conversely, $P_1 \sqcap P_2$ offers an internal choice between the two processes.

The parallel combination $P_1 \underset{A\ B}{\|} P_2$ executes P_1 and P_2 in parallel. P_1 can perform only events in A, P_2 can perform only events in B, and they must simultaneously engage in events in the intersection of A and B.

$P_1 \setminus A$ operation describes the case where all participants of all events in A are described in P_1. All these events are removed from the interface of the process, since no other processes are required to engage in them.

$P_1;\ P_2$ initially executes P_1. When P_1 successfully terminates, then control passes to P_2. This composition can be replicated over a sequence of expressions using the form $;\ x : s @ P$. Finally, the conditional choice if b then P_1 else P_2 behaves as P_1 or P_2 depending on the evaluation of the condition b.

3 Tokeneer: Case Study Introduction

The Tokeneer project [7] is one of the most interesting pilot projects forming part of the Verified Software Initiative [9], and has been cited by the US National Academies as exemplifying best practice in software engineering [10]. The project was certified to Common Criteria Level 5 and in the areas of specification, design

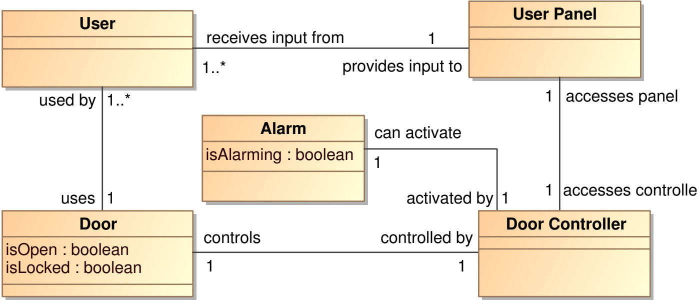


Fig. 2. TIS Class Diagram

and implementation achieving Levels 6 and 7. The Tokeneer project re-developed one component of a Tokeneer system that was developed by the NSA (National Security Agency) to provide protection to secure information held on a network of workstations situated in a physically secure enclave. A survey of other projects using formal methods has been discussed in [11].

To date, only two errors have been found in Tokeneer [7], and the entire project archive has been released [12] for experimentation by researchers. This includes the project specifications written in Z [13] and an open source implementation. Woodcock and Aydal [14] have conducted several experiments using model-based testing techniques to discover twelve anomalous scenarios which challenged the dependability claims for Tokeneer as a security-critical system. Several of the scenarios highlight the importance of the behaviour of the user because one of the security objectives for Tokeneer is to prevent accidental, unauthorised access to the enclave by a user. The user was not formally modelled in the Z specification [12]. We also note the important of the user in our analysis.

Our motivation for using the Tokeneer project as a case study was not to re-validate the project but rather to investigate the concurrent behaviour of the various components of the Tokeneer ID station (TIS) subsystem in the context of asynchronous communication. The correspondence between the Tokeneer formal specifications [12] and our Tokeneer fUML model is not a one-one relationship. Our Tokeneer fUML model contains more implementation details that are abstracted in Tokeneer Z specifications. Therefore, our formal analysis benefits from being able to examine the low level details of asynchronous communication. Such an analysis allows us to investigate potential deadlocks which might occur if the formal specifications were implemented using such communication mechanisms.

The components of interest in the TIS subsystem are represented on the class diagram in Figure 2. We do not formalize the class diagram, and its inclusion is just to illustrate the relationship between the system's components.

***Door*: This is the physical enclave's door that the user opens to access the secure enclave. It has no intelligent behaviour as it is entirely controlled by the door controller component. The two main attributes of this component are: *isOpen* attribute which indicates the status of the door (opened or closed), and the *isLocked* attribute which indicates the status of the door's latch (locked or unlocked).

***Door Controller*:** This component controls the door's latch status (*isLocked*) by setting its value based on the incoming signals from the User Panel. It also manages two timers: the first timer watches if the door is kept closed and unlocked, and the second timer watches if the door is kept opened and locked.

***User*:** This component models the user behaviours toward the system. He is responsible for requesting the enclave entry, and opening the door in case it was successfully unlocked by the User Panel. He is also responsible for closing the door after accessing the enclave. The system may serve more than one user at the same time. However, the results in this paper focus on a single user only.

***User Panel*:** This component models the behaviour of the panel with which the user interfaces to gain access to the enclave. It is responsible for deciding whether the user is allowed to access the enclave or not.

***Alarm*:** This component holds the status of the alarm (alarming or silent), based on the setting/resetting by the Door Controller component to the *isAlarming* attribute.

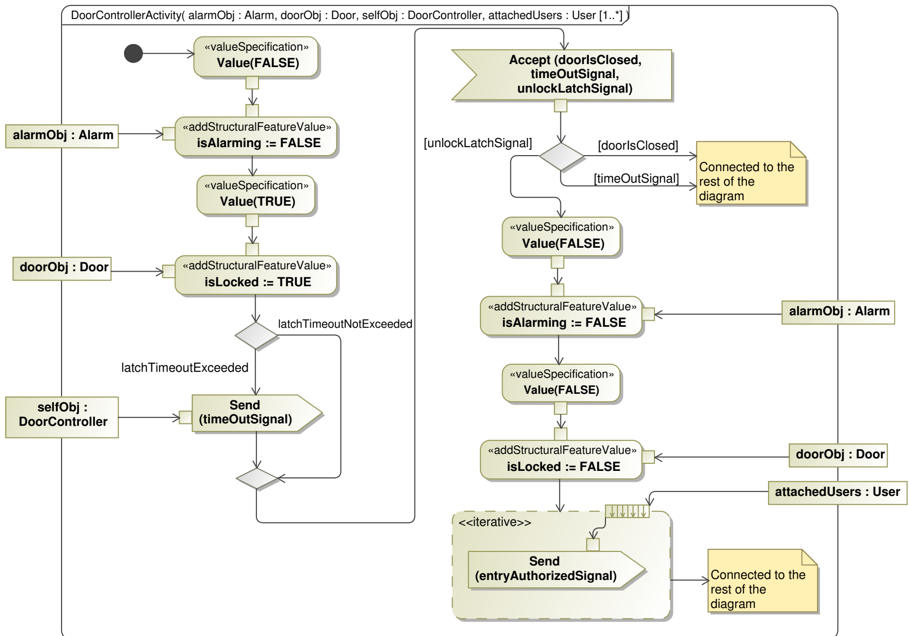


Fig. 3. Segment of the Door Controller Activity

In the Tokeneer fUML model all objects (of the above classes) which have interesting behaviour have associated activity diagrams. The Alarm object is a simple data holder and thus no activity diagram is associated with it. Due to the space constraints, we just focus on a segment of the Door Controller activity (depicted in Figure 3), which includes all the described elements in Section 4. Initially, the Door Controller sets the *isAlarming* variable to FALSE, and *isLocked* to TRUE. At this point the Door's *isOpen* attribute is TRUE, which means the door is open and its latch is locked. For that reason the Door Controller starts a timer to watch this suspicious situation. The activity then represents the two possible scenarios for this timer (*timeOutExceeded*, or *timeOutNotExceeded*). If the timer timeouts the Door Controller sends the *timeOutSignal* to itself to fire the alarm, otherwise it continues with the normal flow. In both cases, the activity waits for one of the following signals to arrive: *closeDoorSignal*, *timeOutSignal*, or *unlockLatchSignal*. If the *unlockLatchSignal* arrives from the User Panel, the Door Controller sets *isLocked* to FALSE, and then sends the *entryAuthorizedSignal* to all Users' objects in the system.

4 Modelling fUML Activity Diagrams into CSP

Table 1 shows the fUML activity diagram's elements and the corresponding CSP representation that reflects the semantic behaviour for each element. As the automatic transformation (from fUML to CSP) is out of this paper's scope, we describe the mapping informally (mapping rules) instead of formally defining transformation rules.

In the mapping rules, *aIH* and *bIH* represent the instance handler of the sender and receiver objects respectively. The values *rp1* and *rp2* represent the registration points on the activity diagram where the object (*bIH*) is waiting to accept the signal instances *sig1* and *sig1*, *sig2*, or *sig3* respectively. Each registration point is unique.

Mapping between UML activity diagrams and CSP has been addressed several times in the literature [15,16]. The novel points of our mapping are as follows:

Rule(1) maps the fUML activity as a parent CSP process that can accept different parameters (*param1*, *param2*, ..). Within this process we define sub-processes, each act as a different fUML element within this activity. The *within* statement defines the action (sub-process) connected to the initial node (*AC1*). Rule(2) and (3) maps the *SendSignalAction* and *AcceptEventAction* to the CSP parameterized events send and accept respectively. The *registerSignals* event is used to let the object activation fill the *waiting event accepters* list with the allowed signals to be accepted at this point (registration point). The value *rp1* is explicitly included in the event so that each *AcceptEventAction* is uniquely identified. Section 5 describes how those events synchronize with the object's buffer process to allow the asynchronous communication between processes (active objects).

Table 1. fUML to CSP Mapping Rules

fUML Element	CSP Representation
Rule(1): Activity P_ACTIVITY(param1, param2) param1 param2	$P_ACTIVITY(param1, param2) =$ let $Activity/Process\ Body$ $within\ AC1$
Rule(2): Send Signal Action bIH bIH Send (sig1)	$AC1 = send!aIH!bIH!sig1 \rightarrow ...$
Rule(3): Accept Event Action Accept(sig1)	$AC1 = registerSignals!bIH!rp1 \rightarrow$ $accept!bIH!sig1 \rightarrow ...$
Rule(4): Accept Event Action (*) Accept(sig1, sig2, sig3) [sig1] [sig3] [sig2]	$AC1 = registerSignals!bIH!rp2 \rightarrow ($ $accept!bIH!sig1 \rightarrow ...$ $\Box$ $accept!bIH!sig2 \rightarrow ...$ $\Box$ $accept!bIH!sig3 \rightarrow ...)$
Rule(5): Add Structural Feature Value Action «valueSpecification» Value(FALSE) aIH «addStructuralFeatureValue» isOpen := FALSE	$AC1 =$ $valueSpec!aIH?value : \{FALSE\} \rightarrow$ $addStFeatureValue!aIH!isOpen!value$ $\rightarrow ...$
Rule(6): Decision/Merge Nodes [decision1] [decision2] Action1 Action2	$DS1 = decision1 \rightarrow AC1$ $\sqcap$ $decision2 \rightarrow AC2$ $AC1 = ... \rightarrow MR1$ $AC2 = ... \rightarrow MR1$ $MR1 = ...$
Rule(7): Expansion Region <<iterative>> Action1	$AC1 = ER_IT1(< e1, e2, e3 >)$ $ER_IT1(<>) = AC2$ $ER_IT1(seq) =;\ s : seq @ Action1!s \rightarrow$ $ER_IT1(tail(seq))$ $AC2 = ...$

The fUML standard supports the fact that the *AcceptEventAction* handles more than one signal at a time. When the control flow of the activity reaches this action, the object waits for any of the defined signals (*sig1*, *sig2*, or *sig3*) to be received. If any of those signals arrive, the object execution proceeds and the incoming signal instance is passed to the *AcceptEventAction* output pin. For that reason, in Rule(4), we connect the decision node to the action's output pin to branch the flow based on the incoming signal. We use the same concept of Rule(3) followed by an external choice to represent the branching semantics. Rules like (2),(3), and (4) are not presented in [15,16] because their focus is not on interaction between activity diagrams.

Rule(5) maps the combination of the actions: *valueSpecificationAction* and *addStructuralFeatureValueAction* to two events to allow (for example) the *aIH* instance handler's attribute *isOpen* to be set to FALSE. We represent the decision node as an internal choice (as in Rule(6)) when the incoming edge to the decision node is a control flow. But we represent it as an external choice (as in Rule(4)) when the incoming edge is an object flow. Having the decision nodes in fUML standard allowed for modelling internal decisions which was not possible using xUML. Rule(7) maps the iterative *ExpansionRegion* as a CSP sequential composition which repeats the action(s) inside the region (*Action1*) with the number of elements inside the sequence *seq*.

Our CSP representation does not include all the properties of the fUML activity diagram elements, as we just focus on the properties in the Tokeneer fUML model. For example, the formalization of the *addStructuralFeatureValueAction* considers the assignment of unordered boolean structural features only.

5 Modelling the fUML Communication Mechanism in CSP

In Section 2.1 we described how active objects in fUML communicate with each others asynchronously and in this section we formalize its semantics using CSP. The model avoids depending on the sequence data structure or Haskell functions, as they lead to a significant decay in FDR performance during the compilation process. For that reason, this implementation uses the CSP primitives only (parallel composition, prefix, etc.). The idea of this implementation came from Michael Goldsmith [17]. As shown in Figure 4, the idea is built on representing the *event pool* as a buffer with N consecutive nodes. When an object sends a signal to another object (perform the *send* event), the signal is placed in the receiver object's buffer (event pool) by placing it in the first node ($B0$), then the signal will move down automatically until reaching the rightmost node in the buffer. The same will be repeated for any other incoming signal filling the buffer from right to left. When the buffer becomes full, the oldest signal in the buffer (placed in the rightmost node) will be dropped out (*drop* event) and all the signals will be shifted right by one node. Signals are moved down as a parameter to the *c1, c2, . . . , cN* events.

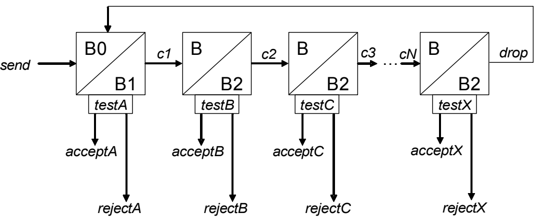
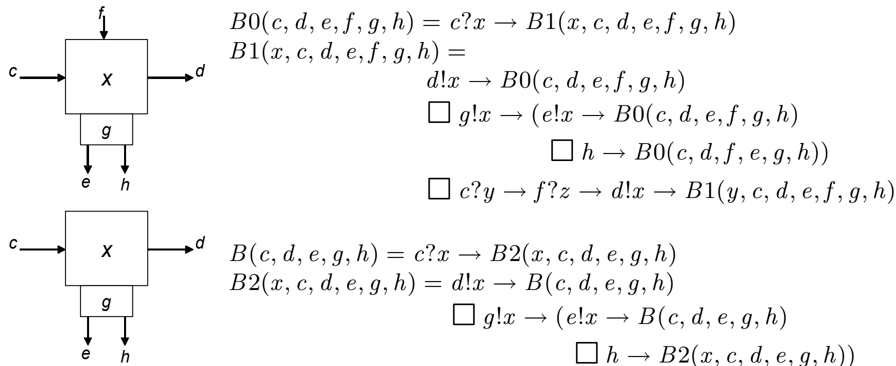


Fig. 4. The Event Pool as a Controlled Buffer

As will be outlined below, the receiver object uses the *testY* event (where Y represents the current node: $A, B, \ldots$) to check if the contained signal is member of the object's *waiting event accepters* list. If so, the signal is removed from the *event pool* via the *acceptY* event, otherwise the *rejectY* event is enabled to allow checking the next node. We represent each of those nodes as a CSP mutually recursive process with a simple logic illustrated in Figure 5 for the first node ($B0$) and the general node (B).

Figure 6 shows the parallel combination of three nodes forming the *event pool* of the Door Controller instance (*dIH0*) which can hold three signal instances at a time. The process *BUF_dcIH* is defined using one *B0* process and two *B* processes whose parameters are instantiated appropriately.



$$
\begin{aligned}
&B0(c, d, e, f, g, h) = c?x \rightarrow B1(x, c, d, e, f, g, h)\\
&B1(x, c, d, e, f, g, h) =\\
&\qquad d!x \rightarrow B0(c, d, e, f, g, h)\\
&\qquad \Box\ g!x \rightarrow (e!x \rightarrow B0(c, d, e, f, g, h)\\
&\qquad\qquad \Box\ h \rightarrow B0(c, d, f, e, g, h))\\
&\qquad \Box\ c?y \rightarrow f?z \rightarrow d!x \rightarrow B1(y, c, d, e, f, g, h)
\end{aligned}
$$

$$
\begin{aligned}
&B(c, d, e, g, h) = c?x \rightarrow B2(x, c, d, e, g, h)\\
&B2(x, c, d, e, g, h) = d!x \rightarrow B(c, d, e, g, h)\\
&\qquad \Box\ g!x \rightarrow (e!x \rightarrow B(c, d, e, g, h)\\
&\qquad\qquad \Box\ h \rightarrow B2(x, c, d, e, g, h))
\end{aligned}
$$

Fig. 5. Buffer's First and General Nodes

$$
\begin{aligned}
BUF_dcIH = ((&B0(send,\ c1,\ acceptA,\ drop,\ testA,\ rejectA)\\
&\underset{\{|c1|\}}{\|}\ B(c1,\ c2,\ acceptB,\ testB,\ rejectB))\\
&\qquad \underset{\{|c2, drop|\}}{\|}\ B(c2,\ drop,\ acceptC,\ testC,\ rejectC)) \setminus \{|\ c1,\ c2,\ drop\ |\}
\end{aligned}
$$

Fig. 6. Three Nodes Controlled Buffer

$$
\begin{aligned}
BUF_CTRL_dcIH(\{\}) = {} & registerSignals!dcIH?rp \rightarrow \\
& \quad BUF_CTRL_dcIH(getRegisteredSignals(dcIH, rp))
\end{aligned}
$$

$$
\begin{aligned}
BUF_CTRL_dcIH(EA) = {} & testC?x \rightarrow if(member(x, EA))\ then \\
& \quad (acceptC!x \rightarrow BUF_CTRL_dcIH(\{\})) \\
& \quad else\ rejectC \rightarrow \\
& testB?x \rightarrow if(member(x, EA))\ then \\
& \quad (acceptB!x \rightarrow BUF_CTRL_dcIH(\{\})) \\
& \quad else\ rejectB \rightarrow \\
& testA?x \rightarrow if(member(x, EA))\ then \\
& \quad (acceptA!x \rightarrow BUF_CTRL_dcIH(\{\})) \\
& \quad else\ rejectA \rightarrow BUF_CTRL_dcIH(EA)
\end{aligned}
$$

Fig. 7. The Buffer Controller Process of the Door Controller Instance

In previous attempts, we have used the default signals dispatching strategy (FIFO) for modelling the inter-object communication. However, this revealed a serious problem when an object receives an unexpected signal (not matched to one of the *waiting event accepters*): the object dismisses it directly because it was removed from its *event pool* for matching and the fUML standard does not allow signals to be returned back to the *event pool*. In many cases the object will need to accept this dismissed signal after some further actions, resulting in a fast invalid deadlock. As the fUML standard allows for overriding the default dispatching strategy, we implemented it by not removing any signal from the *event pool* unless it is registered in its *waiting event accepters* list, to avoid signals dropping. At the same time, signals are dispatched in chronological order (i.e. remove the oldest signal from the *event pool* first) to maintain dispatching signals in the same order they were sent. To meet this logic, we developed a controller process (BUF_CTRL) that checks nodes one by one from the oldest (rightmost) to the newest (leftmost) before removing the signal from the *event pool*, and if the signal exists in the *waiting event accepters* list, the process allows for its acceptance (*accept* event) otherwise the signal is rejected (*reject* event) and the next node is checked. Figure 7 shows our representation of the buffer controller process (BUF_CTRL_dIH) for the Door Controller instance (dIH). The *getRegisteredSignals* is a mapping function that returns the allowed signal(s) at a certain registration point (rp). Note that *registerSignals* event will synchronize with the corresponding event in the translation of the diagram (Rule(3) and (4)).

We depend on the *chase* function of FDR to complete the definition. *Chase* gives priority to internal (tau) transitions over external ones, and chooses one internal transition arbitrarily when there is a choice of several. This is achieved by reducing the state space of the labelled transition system in FDR by removing external transitions competing with internal ones, and selecting one internal transition where there is a choice of them. This results in a refinement of the original process, which can only perform external events once all internal progress has completed. Thus *chase* is not semantics-preserving, but it is exactly what

$$BUF_SYS_dcIH = chase((BUF_dcIH \underset{aSynchEvents}{\parallel} BUF_CTRL_dcIH(\{\})) \setminus aHiddenEvents)$$

Fig. 8. Buffer System Process of the Door Controller Instance

is required here. For more details about how *chase* works the reader can refer to [18].

Figure 8 illustrates the application of *chase* to the parallel combination between the buffer (BUF_dcIH) and the buffer controller (BUF_CTRL_dcIH) of the Door Controller instance after hiding the buffer internal events (*test*, *reject*, *c*, and *drop*) for all nodes (grouped in *aHiddenEvents*). Having those events hidden (*taus*), FDR will follow them causing signals to be propagated along the nodes whenever a *send* event happens. The set *aSynchEvents* contains the synchronization events: *test*, *reject*, and *accept* for all nodes.

The described implementation in this section came after several attempts to model the fUML inter-object communication (semantic variation point). The previous attempts were suffering from various problems like: quick incorrect dropping of signals (which leads to an invalid deadlock), heavy CSP scripts that could not be compiled by FDR, and not maintaining the signals sending order. Having the Tokeneer fUML model formalized in CSP allowed for evaluating those attempts before the actual implementation of the system.

6 Corresponding CSP for Tokeneer fUML Model

Applying the mapping rules of Section 4 to the activity diagram shown in Figure 3 yields the following CSP process in Figure 9. The processes *AC1*, *AC2*, *DS1*, *AC7*, and *AC8* are direct implementation of the mapping rules and they are not involved in the asynchronous communication between the active objects.

The process $DOOR_CTRL_BUF$ represents the parallel combination between the $DOOR_CTRL$ process (represents the Door Controller active object) and its object activation represented formally by the BUF_SYS_dcIH process. When the *send* event happens at *AC3* it synchronizes with the *send* event in the BUF_dcIH to push the *timeOutSignal* inside the Door Controller *event pool* (controlled buffer). After that, the *registerSignals* at *MR1* happens which synchronizes with the *registerSignals* event in the BUF_CTRL_dcIH process, which in turns fills the *waiting event accepters* set (*EA*) with the allowed signals at this point (*doorIsOpenSignal*, *timeOutSignal*, and *unLockLatchSignal*). At this point, the *accept* event in *MR1* synchronizes with the *accept* event in the BUF_CTRL_dcIH process which will happen only if the accepted signal is member of *EA*. The set *aAdcIH* includes all the alphabets of the Door Controller process ($DOOR_CTRL$). The set *aBdcIH* includes the *accept* and *registerSignals* events for all signals the Door Controller accepts, plus the *send* events for any other object sends a signal to the Door Controller process.

$$
\begin{array}{l}
DOOR_CTRL(alarmObj,\ doorObj,\ selfObj,\ attachedUsers) = \\
let \\
\quad AC1 \ = \ valueSpec!selfObj?value : \{FALSE\} \ \rightarrow \\
\qquad\qquad addStFeatureValue!alarmObj!isAlarming!value \rightarrow AC2 \\
\quad AC2 \ = \ valueSpec!selfObj?value : \{TRUE\} \rightarrow \\
\qquad\qquad addStFeatureValue!doorObj!isLocked!value \rightarrow DS1 \\
\quad DS1 \ = \ latchTimeoutExceeded \rightarrow AC3 \sqcap latchTimeoutNotExceeded \rightarrow MR1 \\
\quad AC3 \ = \ send!selfObj!selfObj!timeOutSignal \rightarrow MR1 \\
\quad MR1 \ = \ registerSignals!selfObj!rp1 \rightarrow (accept!selfObj!doorIsClosedSignal \rightarrow ... \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad accept!selfObj!timeOutSignal \rightarrow ... \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad accept!selfObj!unlockLatchSignal \rightarrow AC7) \\
\quad AC7 \ = \ valueSpec!selfObj?value : \{FALSE\} \ \rightarrow \\
\qquad\qquad addStFeatureValue!alarmObj!isAlarming!value \rightarrow AC8 \\
\quad AC8 \ = \ valueSpec!selfObj?value : \{FALSE\} \rightarrow \\
\qquad\qquad addStFeatureValue!doorObj!isLocked!value \rightarrow ER_IT1(attachedUsers) \\
\quad ER_IT1(<>) = ... \\
\quad ER_IT1(users) = ; \quad u : users @ send!selfObj!u!entryAuthorisedSignal \rightarrow \\
\qquad\qquad\qquad ER_IT1(tail(users)) \\
within\ AC1
\end{array}
$$

$$
DOOR_CTRL_BUF = DOOR_CTRL(aIH, dIH, dcIH, uIHS) \underset{aAdcIH \quad aBdcIH}{\|} BUF_SYS_dcIH
$$

Fig. 9. The Corresponding CSP Process for the Door Controller Activity Segment

7 Deadlock Checking

After formalizing the Tokeneer fUML model into CSP, it becomes a direct process to check the behaviour of the model using FDR. In this paper we focus on checking the deadlock that can happen if all the objects in the system are waiting to accept signals from each others. FDR has the capability of checking such a property by determining whether a process (*SYSTEM*) can reach a state in which no further actions are possible, and if a deadlock is found, FDR generates the traces (counter example) that led to this deadlock.

Our *SYSTEM* process includes four interacting processes (Door, Door Controller, User, and User Panel), and each process has its own *event pool* with 10 slots. FDR managed to compile the CSP script (more than 600 lines) in less than a second, and the model checking reported several deadlock scenarios (counter examples). Figure 10 shows part of one of those scenarios represented visually as a sequence diagram to simplify understanding the problem.

Eventually all the objects are waiting for each other causing deadlock. This happens because the User takes a long time (more than the timer period) to open the door after getting the permission to enter from the User Panel. This deadlock was only revealed once we had implemented the asynchronous communication

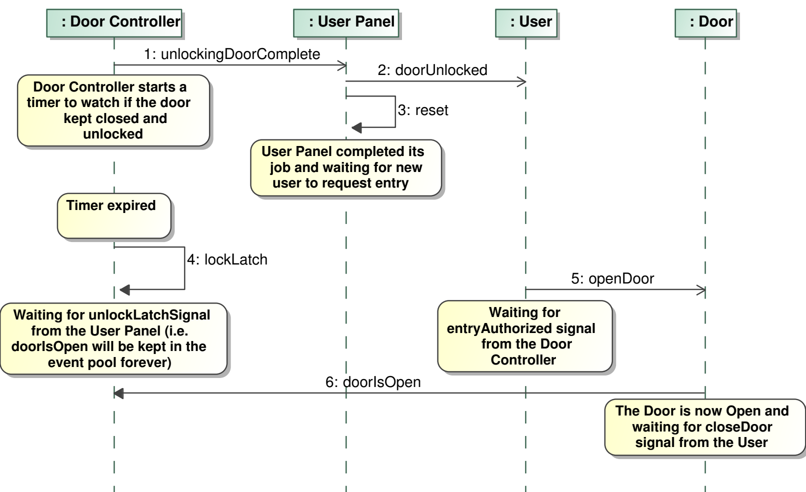


Fig. 10. Sample Scenario Caused a Deadlock

mechanism of Section 5. The deadlock is not a breach of requirements [19] since no explicit timeout is specified in the requirements document which requires a user to enter the enclave within a specific time. However, the Door Controller should not have to wait forever for a User to enter the enclave, therefore it performs a *lockLatch* event after a timer has expired and enters a closed and locked state. In this state the Door Controller never sends the *entryAuthorized* signal to the User because it does not make sense for a User to enter when the door is locked. Consequently, the User cannot evolve its behaviour. Also the *unlockLatch* signal is never sent from the User Panel to the Door Controller and so the Door Controller cannot evolve its behaviour. The deadlock highlights two issues: a naïve model of the User and also the failure of the components to change state during this irregular behaviour. The deadlock we have found demonstrates the importance of modelling all the concurrent interactions that may occur and informs the specifier that implementation issues could cause a deadlock. We would argue that this deadlock was identified because we modelled concurrent behaviour of all the components within the TIS subsystem. The Z specification was not concerned with formalizing the concurrent interaction between components.

During experiments with more primitive communication mechanisms, artificial deadlocks arose because of the inability to process signals. We could not execute the model sufficiently in order to reach the state of Figure 10 because signals were being dropped incorrectly.

8 Related Work

There is a significant body of work relating transforming UML diagrams into formal methods. Among these attempts, some of them, e.g. [20,21,22,23], focus

on formalizing the standard UML diagrams into: Z, Promela, B, or CSP (Constraint Satisfaction Problem). The authors in [20,21] focus on checking consistency between UML diagrams, whereas the authors in [22,24] check refinement between UML models. Translation of UML to CSP has been included in [25,26] which describe how to check the model dynamic behaviours and visualize the formal language into a graphical notation. Our work is more closely compared to [15,16] and [27] which consider the formalization of activity diagrams into CSP, and Petri nets respectively. The authors in [27] focus on checking deadlocks in the UML models, which is aligned to our work. However, none of them addressed modelling the asynchronous communication between objects formally.

Formally representing the asynchronous communication between objects has been discussed in a limited way in [28,29,5] where part of the xUML [4] was formalized, which specify a way of communication different from fUML. On the other hand, [30] simulated the asynchronous message passing by synchronous communication between processes modelling objects and their message queues.

To our knowledge, our work is the first attempt to formalize the fUML standard, and formally represents the asynchronous communication between its active objects. This formal representation provided a formal way to evaluate different implementations (interpretations) of the signals dispatching mechanism (one of the semantic variation points in the fUML standard).

9 Conclusion and Future Work

An approach to model check fUML activity diagrams by representing them in CSP has been presented in this paper. The approach allowed for modelling the inter-object communication (fUML standard semantic variation point) in different ways until reaching a successful implementation. The approach has successfully demonstrated that deadlocks can be detected automatically. Using this approach, the analysis of Tokeneer fUML model CSP representation using FDR, has detected several deadlock scenarios. This means that the fUML model contains some errors, which need to be resolved in the view of the asynchronous communication between objects. Our next step is to investigate this anomaly in the Tokeneer simulator [12].

Using the implementation of the communication mechanism described in Section 5, FDR succeeded in compiling the CSP script in less than a second for a 10 slots *event pool* for each object. Also FDR did not report any dropping of signals using this implementation. However, we do expect signal dropping if the model gets more complicated (e.g. as a result of more than one user), which means identifying the right *event pool* size is very critical to keep the system alive. Currently we do not have a particular methodology to identify the correct object's *event pool* size.

fUML syntax allowed to model all aspects we were interested within Tokeneer. However, we recommend that the value pin be included in the fUML subset, as its exclusion led to a complicated model full of the valueSpecification action.

Currently the transformation from fUML to CSP is done manually based on the mapping rules. In the future, we will automate this transformation using one of the MDA approaches. When we tried to check one of Tokeneer safety specifications we faced a state explosion problem, because FDR generates exponential increasing states when it does the verification. For that reason we may need to represent the fUML model into another formal language and use theorem provers to check such properties.

Acknowledgments. Thanks to Michael Goldsmith and Philip Armstrong for discussion about implementing the buffer in CSP. Thanks also to Ian Wilkie for his helpful information about fUML and to the anonymous referees for their constructive comments.

References

1. OMG: Semantics of a foundational subset for executable UML models (fUML) - (Beta 2) (November 2009), http://www.omg.org/spec/fuml/1.0
2. Schneider, S.: Concurrent and Real-Time Systems: the CSP Approach. Wiley, Chichester (1999)
3. Formal Systems Oxford: FDR 2.83 manual (2007)
4. Mellor, S.J., Balcer, M.J.: Executable UML, A Foundation for Model-Driven Architecture. Addison-Wesley, Reading (2002)
5. Turner, E., Treharne, H., Schneider, S., Evans, N.: Automatic generation of CSP||B skeletons from xUML models. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 364–379. Springer, Heidelberg (2008)
6. Wilkie, I., King, A., Clarke, M., Weaver, C., Raistrick, C., Francis, P.: UML ASL Reference Guide (ASL language level 2.5). Kennedy Carter Ltd. (2003)
7. Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., Everett, B.: Engineering the tokeneer enclave protection software. In: 1st IEEE International Symposium on Secure Software Engineering (March 2006)
8. OMG: Unified modeling language (UML) superstructure (version 2.2) (2009)
9. Hoare, C., Misra, J., Leavens, G.T., Shankar, N.: The verified software initiative: A manifesto. ACM Comput. Surv. 41(4), 1–8 (2009)
10. Johnson, D.: Cost effective software engineering for security. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 607–611. Springer, Heidelberg (2006)
11. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience. ACM Comput. Surv. 41(4), 1–36 (2009)
12. Praxis, A.: The Tokeneer Project, http://www.adacore.com/tokeneer (cited August 2009)
13. Barnes, J., Cooper, D.: Tokeneer ID station: Formal Specification. Technical Report S.P1229.41.2, Altran Praxis (August 2008)
14. Woodcock, J., Aydal, E.G.: A token experiment. Festschrifts in Computer Science, the BCS FAC Series, Festschrift for Tony Hoare (2009)
15. Xu, D., Philbert, N., Liu, Z., Liu, W.: Towards formalizing UML activity diagrams in CSP. In: ISCSCT 2008: Proceedings of the 2008 International Symposium on Computer Science and Computational Technology, Washington, DC, USA, pp. 450–453. IEEE Computer Society, Los Alamitos (2008)

16. Xu, D., Miao, H., Philbert, N.: Model checking UML activity diagrams in FDR. In: ICIS 2009: Proceedings of the 2009 Eigth IEEE/ACIS International Conference on Computer and Information Science, Washington, DC, USA, pp. 1035–1040. IEEE Computer Society, Los Alamitos (2009)
17. Goldsmith, M., Armstrong, P.: Personal communication (February 2010)
18. Zakiuddin, I., Moffat, N., O'Halloran, C., Ryan, P.: Chasing events to certify a critical system. Technical report, UK DERA (1998)
19. Cooper, D., Barnes, J.: Tokeneer ID station: System Requirements Specification. Technical Report S.P1229.41.1, Altran Praxis (August 2008)
20. Amalio, N., Stepney, S., Polack, F.: Formal proof from UML models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 418–433. Springer, Heidelberg (2004)
21. Zhao, X., Long, Q., Qiu, Z.: Model checking dynamic UML consistency. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 440–459. Springer, Heidelberg (2006)
22. Ammar, B.B., Bhiri, M.T., Souquières, J.: Incremental development of UML specifications using operation refinements. ISSE 4(3), 259–266 (2008)
23. Cabot, J., Clarisó, R., Riera, D.: Verifying UML/OCL operation contracts. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 40–55. Springer, Heidelberg (2009)
24. Pons, C.: Heuristics on the definition of UML refinement patterns. In: Wiedermann, J., Tel, G., Pokorný, J., Bieliková, M., Štuller, J. (eds.) SOFSEM 2006. LNCS, vol. 3831, pp. 461–470. Springer, Heidelberg (2006)
25. Ng, M.Y., Butler, M.J.: Tool support for visualizing CSP in UML. In: George, C.W., Miao, H. (eds.) ICFEM 2002. LNCS, vol. 2495, pp. 287–298. Springer, Heidelberg (2002)
26. Ng, M.Y., Butler, M.: Towards formalizing UML state diagrams in CSP. In: Cerone, A., Lindsay, P. (eds.) 1st IEEE International Conference on Software Engineering and Formal Methods, pp. 138–147. IEEE Computer Society, Los Alamitos (2003)
27. Thierry-Mieg, Y., Hillah, L.M.: UML behavioral consistency checking using instantiable Petri nets. ISSE 4(3), 293–300 (2008)
28. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M., van de Pol, J.: Towards model checking Executable UML specifications in mCRL2. ISSE, 83–90 (2010)
29. Graw, G., Herrmann, P.: Transformation and verification of Executable UML models. Electron. Notes Theor. Comput. Sci. 101, 3–24 (2004)
30. Xie, F., Levin, V., Browne, J.C.: Model checking for an executable subset of UML. In: ASE 2001: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, p. 333. IEEE Computer Society, Los Alamitos (2001)

Model Checking Hierarchical Probabilistic Systems*

Jun Sun[1], Songzheng Song[3], and Yang Liu[2]

[1] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] National University of Singapore
liuyang@comp.nus.edu.sg
[3] NUS Graduate School for Integrative Sciences and Engineering
songsongzheng@nus.edu.sg

Abstract. Probabilistic modeling is important for random distributed algorithms, bio-systems or decision processes. Probabilistic model checking is a systematic way of analyzing finite-state probabilistic models. Existing probabilistic model checkers have been designed for simple systems without hierarchy. In this paper, we extend the PAT toolkit to support probabilistic model checking of hierarchical complex systems. We propose to use PCSP#, a combination of Hoare's CSP with data and probability, to model such systems. In addition to temporal logic, we allow complex safety properties to be specified by non-probabilistic PCSP# model. Validity of the properties (with probability) is established by refinement checking. Furthermore, we show that refinement checking can be applied to verify probabilistic systems against safety/co-safety temporal logic properties efficiently. We demonstrate the usability and scalability of the extended PAT checker via automated verification of benchmark systems and comparison with state-of-art probabilistic model checkers.

1 Introduction

Probabilistic systems are common in practice, e.g., randomized algorithms, unreliable system components, unpredictable environment, etc. Probabilistic model checking is a systematic way of analyzing finite-state probabilistic systems. Given a finite-state model of a probabilistic system and a property, a probabilistic model checker calculates the (range of) probability that the model satisfies the property. It has been proven useful in a variety of domains (see examples in [14]).

Designing and verifying probabilistic systems is becoming an increasingly difficult task due to the widespread applications and increasing complexity of such systems. Existing probabilistic model checkers have been designed for hierarchically simple systems. For instance, the popular PRISM checker [14] supports a simple state-based language, based on the Reactive Modules formalism of Alur and Henzinger [2]. The MRMC checker supports a rather simple input language too [16]. The input language of the LiQuor checker [10], named Probmela, is based on an extension of Promela supported by the SPIN model checker. None of the above checkers supports analysis of hierarchical complex probabilistic systems.

* This research was partially supported by a grant "SRG ISTD 2010 001" from Singapore University of Technology and Design.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 388–403, 2010.
© Springer-Verlag Berlin Heidelberg 2010

In this work, we aim to develop a useful tool for verifying hierarchical complex probabilistic systems. Firstly, we propose a language called PCSP# for system modeling. PCSP# is an expressive language, combining Hoare's CSP [15], data structures, and probabilistic choices. It extends previous work on combining CSP with probabilistic choice [22] or on combining CSP with data structures [30]. PCSP# combines low-level programs, e.g., sequence programs defined in a simple imperative language or any C# program, with high-level specifications (with process constructs like parallel, choice, hiding, etc.), as well as probabilistic choices. It supports shared variables as well as abstract events, making it both state-based and event-based. Its underlying semantics is based on Markov Decision Processes (MDP) [6].

Secondly, we propose to verify complex safety properties by showing a refinement relationship (with probability) from a PCSP# model representing a system and a non-probabilistic model representing properties. Note that we assume that the property model is non-probabilistic. We view probability as a necessary devil forced upon us by the unreliability of the system or its environment. In contrast, properties which characterizes correct system behaviors are often irrelevant of the likelihood of some low-level failures. Refinement checking has been traditionally used to verify variants of CSP [26,27]. It has been proven useful by the success of the FDR checker [27]. Verification of such properties are reduced to the problem of probabilistic model checking against deterministic finite automata, which has been previously solved (see for example [4]). Nonetheless, we present a slightly improved algorithm which is better suited for our setting. Alternatively, properties can be stated in form of state/event linear temporal logic (SE-LTL) [8]. An SE-LTL formula can be built from not only atomic state propositions but also events, making it a perfect property specification language for PCSP#, which is both state-based and event-based. A standard method for model checking SE-LTL formulae is the automata-based approach [4]. In this paper, we improve it by safety/co-safety recognition. That is, if an LTL formula or its negation is recognized as a safety property, then the model checking problem is reduced to a refinement checking problem and solved using our refinement checking algorithm. Though the worse-case complexity remains the same, we show that safety/co-safety recognition offers significantly memory/time saving in practice. Lastly, we extend the PAT model checker with all the techniques so as to offer a self-contained framework for probabilistic system modeling, simulation (using the built-in visualized simulator), and verification. In order to demonstrate the usability/scalability of our approach, we verify benchmark systems and compare the results with the PRISM checker [14].

Related work. This work is related to methods and tools for probabilistic system modeling and verification. Existing probabilistic model checkers include at least PRISM [14], MRMC [16] and LiQuor [10]. PRISM is the most popular probabilistic model checker. It supports a variety of probabilistic models as well as property specification languages. The input of PRISM is a simple state-based language [2]. LiQuor is a probabilistic model checker for reactive systems [10]. MRMC is a command-line based model checker for a variety of probabilistic models and a rather simple input language. The extended PAT checker complements the existing checkers by 1) offering a language that is both state-based and event-based and is capable of modeling hierarchical systems;

2) supporting both SE-LTL model checking and probabilistic refinement checking and 3) offering a user-friendly environment for not only model checking but also simulation.

The language PCSP# is related to many works on integrating probabilistic behaviors into process algebras or programs, among which the most relevant are [22,21,9,33]. In [22], an extension of CSP is proposed to incorporate probabilistic behaviors in the name of refinement checking. In [21], issues on integrating probability with Event-B has been discussed. In [9], issues on integrating probability with non-determinism have been addressed. Compared to [22,21,9,33], this work focuses on developing a practical tool for systematic modeling and verification of probabilistic systems.

Our work on improving temporal logic model checking with safety recognition is related to work on categorizing safety and liveness. The work presented in [1] offers theoretical results for recognizing safety and liveness given a Büchi automaton. Others have also considered the problem of model checking safety LTL properties. In [28], a categorization of safety, liveness and fairness is discussed. Further, it showed that recognizing safety LTL properties is PSPACE-complete. Later, many theoretical results and algorithms have been presented in [17], which generalizes the earlier work presented in [28]. A forward direction version of the algorithm in [17] is evidenced in [12]. In [18], the author presented a translation of safety LTL formula to a finite state automaton which detects bad prefixes. Model checking safety properties expressed using past temporal operators has been considered in [13]. Our safety recognition is based on [1,28]. Different from the above, we present methods/algorithms which improve model checking of not only safety properties but also a class of liveness properties; not only finite state systems but also probabilistic systems.

Organization. The remainder of the paper is organized as follows. Section 2 presents relevant technical definitions. Section 3 introduces the syntax and semantics of PCSP#. Section 4 presents probabilistic verification of PCSP# models. In particular, Section 4.1 presents a refinement checking algorithm. Section 4.2 presents our approach for verifying SE-LTL formulae with safety recognition. Section 5 evaluates our methods. Section 6 concludes the paper with future research directions.

2 Preliminaries

LTS. A labeled transition system (LTS) $\mathcal{L}$ is a tuple $(S, init, Act, T)$ where S is a finite set of states; and $init \in S$ is an initial state; Act is an alphabet; $T \subseteq S \times Act \times S$ is a labeled transition relation. A transition label can be either a visible event or an invisible one (which is referred to as τ). A τ-transition is a transition labeled with τ. For simplicity, we write $s \xrightarrow{e} s'$ to denote $(s, e, s') \in T$. If $s \xrightarrow{e} s'$, then we say that e is enabled at s. Let $s \rightsquigarrow s'$ to denote that s' can be reached from s via zero or more τ-transitions; we write $s \overset{e}{\rightsquigarrow} s'$ to denote there exists s_0 and s_1 such that $s \rightsquigarrow s_0 \xrightarrow{e} s_1 \rightsquigarrow s'$. A path of $\mathcal{L}$ is a sequence of alternating states/events $\pi = \langle s_0, e_0, s_1, e_1, \cdots \rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} s_{i+1}$ for all i. The set of path of $\mathcal{L}$ is written as $paths(\mathcal{L})$. Given a path π, we can obtain a sequence of visible events by omitting states and τ-events. The sequence, written as $trace(\pi)$, is a trace of $\mathcal{L}$. The set of traces of $\mathcal{L}$ is written as $traces(\mathcal{L}) = \{trace(\pi) \mid \pi \in paths(\mathcal{L})\}$. An LTS is deterministic if and only if given

any s and e, there exists only one s' such that $s \xrightarrow{e} s'$. An LTS is non-deterministic if and only if it is not deterministic. A non-deterministic LTS can be translated into a trace-equivalent deterministic LTS by determinization. Furthermore, non-deterministic LTSs containing τ-transitions can be translated into trace-equivalent deterministic LTSs without τ-transitions. The process is known as normalization [26].

Definition 1 (Normalization). *Let $\mathcal{L} = (S, init, Act, T)$ be an LTS. The normalized LTS of $\mathcal{L}$ is $nl(\mathcal{L}) = (S', init', Act, T')$ where $S' \subseteq 2^S$ is a set of sets of states, $init' = \{s \mid init \rightsquigarrow s\}$ and T' is a transition relation satisfying the following condition: $(N, e, N') \in T'$ if and only if $N' = \{s' \mid \exists\, s : N.\ s \overset{e}{\rightsquigarrow} s'\}$.*

Normalization is to group states which can be reached via the same trace. Given two LTSs $\mathcal{L}_0$ and $\mathcal{L}_1$, it is often useful to check whether $traces(\mathcal{L}_0)$ is a subset of $traces(\mathcal{L}_1)$ (or equivalently $\mathcal{L}_0$ trace-refines $\mathcal{L}_1$). There are existing algorithms and tools for trace inclusion check [26]. The idea is to construct the product of $\mathcal{L}_0$ and $nl(\mathcal{L}_1)$ and then search for a state of the form (s, s') such that s enables more visible events than s' does. In the worse case, this algorithm is exponential in the number of states of $\mathcal{L}_1$. It is nonetheless proven to be practical for real-world systems by the success of the FDR checker [27].

SE-LTL. LTL was introduced to specify the properties of executions of a system [24]. It is built up from a set of *propositions* using standard Boolean operators ($\neg$, $\wedge$, $\vee$) and X (next), U (until), R (release), $\Diamond$ (eventually) and $\Box$ (always). It has been adopted for specifying properties in many systems. In [8], LTL is extended to build up from not only state propositions but also events. The extended LTL is referred to as SE-LTL. The simplicity of writing formulas concerning events as in the above example is not purely a matter of aesthetics. It may yield gains in time and space [8].

LTL (and SE-LTL) formulae can be categorized into either safety or liveness. Informally speaking, safety properties stipulate that "bad things" do not happen during system execution. A finite execution is sufficient evidence to the violation of a safety property. In contrast, liveness properties stipulate that "good things" do happen eventually. A counterexample to a liveness property is an infinite system execution (which forms a loop if the system has finitely many states). In this paper, we adopt the definition of safety and liveness in [1]. For instance, $\Box(a \Rightarrow \Box b)$ and $\Diamond a \Rightarrow \Box b$ are safety properties; $\Box a \Rightarrow \Diamond b$ is a liveness property, whose negation, however, is a safety property. A liveness property whose negation is safety is referred to as co-safety, e.g., $\Diamond a$ is co-safety. We remark that a formula may be neither safety nor liveness, e.g., $\Box\Diamond a \wedge \Box b$. It has been shown in [28] that recognizing whether an LTL formula is safety is PSPACE-compele. A number of methods have been proposed to identify subsets of safety. For instance, *syntactic LTL safety formulae* (which is constituted by $\wedge$, $\vee$, $\Box$, U, X, and propositions or negations of propositions) can be recognized efficiently. A number of methods have been proposed to translate safety LTL to finite state automata [17,18].

It has been proved in [32] that for every LTL formula ϕ, there exists an equivalent Büchi Automaton. There are many sophisticated algorithms on translating LTL to an equivalent Büchi automaton [11,29]. In addition, it is possible to tell whether an LTL

formula represents safety by examining its equivalent Büchi automaton. For instance, it has been proved in [1] that a (reduced) Büchi automaton specifies a safety property if and only if making all of its states accepting does not change its language. Based on this result, *a Büchi automaton representing a safety property can be viewed as an LTS for simplicity*. The reason is that all of its infinite traces must be accepting and therefore the acceptance condition can be ignored.

3 Hierarchical Modeling

In this section, we present PCSP#, which is designed for modeling and verifying probabilistic systems. We remark that the LiQuor checker, which is based on Probmela, makes a step towards an expressive useful modeling language. Nonetheless, Probmela is not capable of modeling fully hierarchical systems.

Syntax. PCSP# extends the CSP# language [30] with probabilistic choices. CSP# integrates low-level programs with high-level compositional specification. It is capable of modeling systems with not only complicated data structures (which are manipulated by the low-level programs) but also hierarchical systems with complex control flows (which are specified by the high-level specification). Compared with PCSP [22], PCSP# supports explicit complex data structures/operations.

A PCSP# model is a 3-tuple $(Var, init, P)$ where Var is a set of global variables (with bounded domains) and channels; $init$ is the initial values of Var; P is a process. A variable can be either of simple types like boolean, integer, arrays of integers or any user-defined data type (which must be defined in an external C# library). The process P is an extension of Hoare's classic CSP. Part of its syntax is defined as follows.

$$\begin{array}{lll}
P ::= & Stop \mid Skip & \text{– primitives} \\
\mid & e \rightarrow P & \text{– event prefixing} \\
\mid & a\{program\} \rightarrow P & \text{– data operation prefix} \\
\mid & P \Box Q \mid P \sqcap Q \mid \textbf{if } b \textbf{ then } P \textbf{ else } Q & \text{– choices} \\
\mid & P;\ Q & \text{– sequence} \\
\mid & P \parallel Q \mid P \mid\mid\mid Q & \text{– concurrency} \\
\mid & P \setminus X & \text{– hiding} \\
\mid & Q & \text{– process referencing} \\
\mid & \textbf{pcase}\ \{d_0 : P_0;\ d_1 : P_1;\ \cdots;\ d_k : P_k\} & \text{– probabilistic multi-choices}
\end{array}$$

where P, P_i and Q range over processes, e is a simple event, a is the name of a sequential program; b is a Boolean expression, d_i is a rational number and $d_0 + d_1 + \cdots + d_k = 1$. Process $Stop$ does nothing. Process $Skip$ terminates. Process $e \rightarrow P$ engages in event e first and then behaves as P. Combined with parallel composition, event e may serve as a multi-party synchronization barrier. Process $a\{program\} \rightarrow P$ generates an event a, executes a sequential program $program$ at the same time, and then behaves as P. External C# data operations can be invoked in $program$.

A variety of choices are supported, e.g., $P \Box Q$ for external choice; $P \sqcap$ for internal non-determinism and **if** b **then** P **else** Q for conditional branching. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q. Parallel composition of two

processes is written as $P \parallel Q$, where P and Q may communicate via multi-party event synchronization. If P and Q only communicate through channels or variables, then it is written as $P \mid\mid\mid Q$. Process $P \setminus X$ hides occurrence of any event in X. Recursion is supported through process referencing. Lastly, probabilistic choice is written in the form of **pcase** $\{d_0 : P_0;\ d_1 : P_1;\ \cdots;\ d_k : P_k\}$. Intuitively, it means that with d_i probability, the system behaves as P_i. It is required that $d_0 + d_1 + \cdots + d_k = 1$.

Example 1 (Pacemaker). A pacemaker is an electronic implanted device which functions to regulate the heart beat by electrically stimulating the heart to contract and thus to pump blood throughout the body. Common pacemakers are designed to correct bradycardia, i.e., slow heart beats. A pacemaker mainly performs two functions, i.e., sensing and pacing. Sensing is to monitor the heart's natural electrical activity, helping the pacemaker to gather information on the heart beats and react accordingly. Pacing is when a pacemaker sends electrical stimuli, i.e., tiny electrical signals, to heart through a pacing lead, which starts a heart beat. A pacemaker can operate in many different modes, according to the implanted patient's heart problem. The following is a high-level abstraction of the simplest mode of pacemaker, i.e., the AAT mode.

$$
\begin{array}{ll}
\textbf{var}\ count & = 0;\\
AAT & = (Heart \parallel Pacing) \setminus \{missingPulseA, missingPulseV\}\\
Heart & = \textbf{pcase}\ \{\\
& \qquad [pA] : missingPulseA \rightarrow pulseV \rightarrow Heart\\
& \qquad [pV] : pulseA \rightarrow missingPulseV \rightarrow Heart\\
& \qquad [1 - pA - pV] : pulseA \rightarrow pulseV \rightarrow Heart\\
& \};\\
Pacing & = pulseA \rightarrow Pacing \ \Box\ pulseV \rightarrow Pacing\\
& \Box\ missingPulseA \rightarrow add\{count++\} \rightarrow \textbf{pcase}\ \{\\
& \qquad\qquad [99.54] : pulseB\{count--\} \rightarrow Pacing\\
& \qquad\qquad [0.46] : Pacing\\
& \}\\
& \Box\ missingPulseV \rightarrow add\{count++\} \rightarrow \textbf{pcase}\ \{\\
& \qquad\qquad [99.54] : pulseW\{count--\} \rightarrow Pacing\\
& \qquad\qquad [0.46] : Pacing\\
& \};
\end{array}
$$

Variable $count$ is an integer (with a default bound) which records the number of skipped pulses. A (mode of the) pacemaker is typically modeled in the following form: $Heart \parallel Pacing$ where $Heart$ models normal or abnormal heart condition; $Pacing$ models how the pacemaker functions. In this particular mode, process $Heart$ generates two events $pulseA$ (i.e., atrium does a pulse) and $pulseV$ (i.e., ventricle does a pulse), periodically for a normal heart or with one of them missing once a while for an abnormal heart. In the latter case, event $missingPulseA$ or $missingPulseV$ is generated. Constant pA is the (patient-dependent) probability of $pulseA$ missing; pV is the probability of $pulseV$ missing. Process $Pacing$ synchronizes with process $Heart$. If event $missingPulseA$ (denoting the missing of event $pulseA$) is monitored, variable $count$ is incremented by one. Notice that the event add is associated with the simple program of updating $count$. In general, it can be associated with any state update function. It is in this way that state

update is introduced in an event-based language. Ideally, the pacemaker helps the heart to beat by generating event $pulseB$. Once $pulseB$ is generated, $count$ is decremented by one. Similarly, it generates $pulseW$ when $pulseV$ is missing. Note that it has been reported that pacemaker may malfunction for certain rate (exactly 0.46%) [20]. This is reflected in the model again using **pcase**. If a $pulseB$ or $pulseW$ is skipped, $count$ is not decremented.

At the top level, the pacemaker system is a choice of different modes. Each mode is often a parallel composition of multiple components. Each component may have internally hierarchies due to complicated sensing and pacing behaviors. We skip the details (refer to [5]) and remark that our modeling language is more suitable for such systems than those supported by existing probabilistic model checkers. □

Semantics. The semantic model of CSP# (without **pcase**) is LTS. In this paper, we assume that all variables have finite domain and the set of reachable process expressions are finite so that the LTS has finitely states. A state in the LTS is a tuple of the form (V, P) where V is the valuation of the variables and P is a process expression. Given a CSP# model $\mathcal{M}$, its LTS can be generated systematically following its structural operational semantics (also known as firing rules [30]). A firing rule for CSP# is of the form $(V, P) \xrightarrow{e} (V', P')$. Based on the LTS, different semantic objects can be defined. For instance, the traces of $\mathcal{M}$ are defined to be the traces of the LTS.

The underlying semantics of PCSP# is Markov Decision Process (MDP), which is expressive enough to capture systems with probabilistic choices as well as nondeterminism and concurrency. Given a set of states S, a distribution is a function $\mu : S \rightarrow [0, 1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S. An MDP is a 3-tuple $\mathcal{M} = (S, init, Pr)$ where S is a set of system states; $init \in S$ is the initial system configuration[1]; $Pr : S \times Act \times Distr(S)$ is a transition relation[2]. A transition of the system is written as $s \xrightarrow{e} \mu$ where μ is a distribution, or equivalently $s \xrightarrow{e} \{(s_1, d_1), (s_2, d_2), \cdots\}$ where $s \in S$ and $s_i \in S$ for all i; and $d_i : [0, 1]$ is the probability of reaching s_i given the distribution. A path of $\mathcal{M}$ is a sequence of alternating states, events and distributions $\pi = \langle s_0, e_0, \mu_0, s_1, e_1, \mu_1, \cdots\rangle$ such that $s_0 = init$ and $s_i \xrightarrow{e_i} \mu_i$ and $\mu_i(s_{i+1}) > 0$ for all i. The probability of exhibiting π by $\mathcal{M}$, denoted as $\mathcal{P}_{\mathcal{M}}(\pi)$, is $\mu_0(s_1) * \mu_1(s_2) * \cdots$. Given a path π, we define $trace(\pi)$ to be the sequence of visible events in π. Let $paths(\mathcal{M})$ denote all paths of $\mathcal{M}$. In an abuse of notation, let $s \in \pi$ denote that π visits state s.

In the following, *we assume that MDPs are deadlock-free* following common practice. A deadlocking model can be made deadlock-free by adding a special self loop to the deadlock states, without affecting the result of probabilistic verification. Intuitively speaking, given a system configuration, firstly an event and a distribution is selected nondeterministically by the scheduler, and then one of successor states is reached according to the probability distribution. A scheduler is a function decides which event and distribution to choose based on the execution history (in the form of a path). A Markov Chain [4] can be defined given an MDP $\mathcal{M}$ and a scheduler δ, denoted as $\mathcal{M}_\delta$. Intuitively, a Markov Chain is an MDP where only one event and

[1] This is a simplified definition. In general, there can be an initial distribution.

[2] This is slightly different from the classic definition of MDP.

Rule for any process constructs in CSP#:

$$\frac{(V, P) \xrightarrow{e} (V', P') \text{ is a firing rule of CSP\#}}{(V, P) \xrightarrow{e} \mu \text{ such that } \mu((V', P')) = 1}$$

Rule for **pcase**

$$\frac{}{(V, \textbf{pcase}\ \{d_0 : P_0;\ \cdots;\ d_k : P_k\}) \xrightarrow{\tau} \mu \text{ such that } \mu((V, P_i)) = d_i \text{ for all } i}$$

Fig. 1. Firing Rules

distribution is enabled at every state. For simplicity, we write $\mathcal{P}^{\delta}_{\mathcal{M}}(\pi)$ to denote the probability of exhibiting a path π in $\mathcal{M}$ with scheduler δ. The probability of exhibiting a set of path $X \subseteq paths(\mathcal{M}_\delta)$ is the accumulated probability of each path, i.e., $\mathcal{P}^{\delta}_{\mathcal{M}}(X) = \Sigma_{x \in X} \mathcal{P}^{\delta}_{\mathcal{M}}(x)$. It is often useful to find out the probability of reaching a set of states. Note that with different scheduling, the probability may be different. The measurement of interest is thus the maximum and minimum probability. Given a set of target states G, the maximum probability of reaching any state in G is defined as

$$\mathcal{P}^{max}_{\mathcal{M}}(G) = \sup_\delta Pr^{\delta}_{\mathcal{M}}(\{\pi \mid \exists\, s \in G.\ s \in \pi\})$$

Note that the supremum ranges over all, potentially infinitely many, schedulers. Accordingly, the minimum is written as $\mathcal{P}^{min}_{\mathcal{M}}(G)$. Similarly, we define the maximum probability of exhibiting a trace in a set Tr by $\mathcal{M}$.

$$\mathcal{P}^{max}_{\mathcal{M}}(Tr) = \sup_\delta Pr^{\delta}_{\mathcal{M}}(\{\pi \mid trace(\pi) \in Tr\})$$

Accordingly, the minimum is written as $\mathcal{P}^{min}_{\mathcal{M}}(Tr)$.

Next, we define firing rules for PCSP#. Figure 1 presents all the necessary rules. The first rule states that if by CSP# firing rules, $(V, P) \xrightarrow{e} (V', P')$, then configuration (V, P) can perform e and result in one distribution which maps configuration (V', P') to 1. The second rule is for **pcase**. The result distribution associates d_k probability with (V, P_k) for all k. Notice that V remains unmodified during the transition. We remark that only τ-transitions can be associated with probability other than 1. Following these two rules, an MDP can be generated from a model systematically.

4 Probabilistic Refinement Checking

Refinement checking has been traditionally used to verify CSP [15]. Different from temporal-logic based model checking, refinement checking works by taking a model (often in the same language) as a property. The property is verified by showing a refinement relationship from the system model to the property model. There are different refinement relationships designed for proving different properties. In the following, we focus on trace refinement and remark that our approach can be extended to stable failures refinement or failures/divergence refinement. For instance, one way of verifying

the pacemaker is to check whether the pacemaker model (present in Example 1) trace-refines the following model (without variables) which models a 'fine' heart.

$$OKHrt = pulseA \rightarrow pulseV \rightarrow OKHrt \;\Box\; pulseA \rightarrow pulseW \rightarrow OKHrt \;\Box\; pulseB \rightarrow pulseV \rightarrow OKHrt \;\Box\; pulseB \rightarrow pulseW \rightarrow OKHrt$$

In theory, it is possible to encode the property model as temporal logic formulae (as temporal logic is typically more expressive than LTS) and then apply temporal-logic based model checking to verify the property. It is, however, impractical. For instance, LTL model checking is exponential in the size of the formulae and therefore it cannot handle formulae which encode non-trivial property model. In short, refinement checking allows users to verify a different class of properties from temporal logic formulae.

4.1 Refinement Checking PCSP#

Because of probabilistic choices, refinement checking in our setting is not simply to verify whether traces of a PCSP# model is subset of those of another. Instead, it is 'how likely' the system behaves as specified by the property model (in the presence of unreliability of system components). Because we assume the property model is non-probabilistic, the problem is thus to calculate the probability of an MDP (i.e., the semantics of PCSP# model) trace-refines an LTS (i.e., the semantics of a non-probabilistic PCSP# model).

Definition 2 (Refinement Probability). *Let $\mathcal{M}$ be an MDP and $\mathcal{L}$ be an LTS. The maximum probability of $\mathcal{M}$ trace-refines $\mathcal{L}$ is defined by $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L}) = \mathcal{P}^{max}_{\mathcal{M}}(traces(\mathcal{L}))$. The minimum is defined by $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L}) = \mathcal{P}^{min}_{\mathcal{M}}(traces(\mathcal{L}))$.* □

Intuitively, the probability of $\mathcal{M}$ refines $\mathcal{L}$ is the sum of the probability of $\mathcal{M}$ exhibiting every trace of $\mathcal{L}$. The probability may vary due to different scheduling. One way of calculating the maximum/minimum probability [4] is to (1) build a deterministic LTS $\mathcal{L}^{-1}$ which complements $\mathcal{L}$ (such that $traces(\mathcal{L}^{-1}) = \Sigma^* \setminus traces(\mathcal{L})$); (2) compute the product of $\mathcal{M}$ and $\mathcal{L}^{-1}$; 3) calculate the maximum/minimum probability of paths of the product.

In the following, we present a slightly improved algorithm which avoids the construction of $\mathcal{L}^{-1}$. Note that for a complicated language like PCSP#, computing $\mathcal{L}^{-1}$ is highly nontrivial. The algorithm is inspired by the refinement checking algorithm in FDR. Firstly, we normalize $\mathcal{L}$ using the standard powerset construction. Next, we compute the synchronous product of $\mathcal{M}$ and $nl(\mathcal{L})$, written as $\mathcal{M} \times nl(\mathcal{L})$. It can be shown that the product is an MDP.

Definition 3 (Product MDP). *Let $\mathcal{M} = (S_{\mathcal{M}}, init_{\mathcal{M}}, Act, Pr_{\mathcal{M}})$ be an MDP and $\mathcal{L} = (S_{\mathcal{L}}, init_{\mathcal{L}}, Act, T_{\mathcal{L}})$ be a deterministic LTS without τ-transitions. The product is the MDP $\mathcal{M} \times \mathcal{L} = (S_{\mathcal{M}} \times S_{\mathcal{L}}, (init_{\mathcal{M}}, init_{\mathcal{L}}), Act, Pr)$ such that Pr is the least transition relation which satisfies the following conditions.*

- *If $s_m \xrightarrow{\tau} \mu$ in $\mathcal{M}$, then $(s_m, s_l) \xrightarrow{\tau} \mu'$ in $\mathcal{M} \times \mathcal{L}$ for all $s_l \in S_{\mathcal{L}}$ such that $\mu'((s'_m, s_l)) = \mu(s'_m)$ for all $s'_m \in S_{\mathcal{M}}$.*

– *If* $s_m \xrightarrow{e} \mu$ *in* $\mathcal{M}$ *and* $s_l \xrightarrow{e} s_l'$ *in* $\mathcal{L}$*, then* $(s_m, s_l) \xrightarrow{e} \mu'$ *in* $\mathcal{M} \times \mathcal{L}$ *such that* $\mu'((s_m', s_l')) = \mu(s_m')$ *for all* $s_m' \in S_{\mathcal{M}}$*.*

In the product, there are two kinds of transitions, i.e., τ-transitions from $\mathcal{M}$ with the same probability or transitions labeled with a visible event with probability 1. Note that τ-transitions are not synchronized, whereas visible events must be jointly performed by $\mathcal{M}$ and $\mathcal{L}$. Let $G \subseteq S_{\mathcal{M}} \times S_{\mathcal{L}}$ be the least set of states satisfying the following condition: for every pair $(s, s') \in G$, $s' = \varnothing$. Intuitively, $(s, s') \in G$ if and only if a trace of $\mathcal{M}$ leading to s is not possible in $\mathcal{L}$. The following theorem states our main result on refinement checking.

Theorem 1. *Let* $\mathcal{M}$ *be an MDP;* $\mathcal{L}$ *be an LTS;* $\mathcal{D} = \mathcal{M} \times nl(\mathcal{L})$*.* $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L}) = 1 - \mathcal{P}^{min}_{\mathcal{D}}(G)$ *and* $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L}) = 1 - \mathcal{P}^{max}_{\mathcal{D}}(G)$*.*

Proof. Let δ be any scheduler for $\mathcal{M}$. Note that δ can be extended to be a scheduler for $\mathcal{D}$ straightforwardly. For simplicity, we use δ to denote both of them. Let $X \subseteq paths(\mathcal{M})$. The following shows that the equivalence holds with any scheduler.

$$\begin{array}{ll}
\mathcal{P}^{\delta}_{M}(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \in traces(\mathcal{L})\}) & \\
\equiv 1 - \mathcal{P}^{\delta}_{M}(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \notin traces(\mathcal{L})\}) & \text{– by def.} \\
\equiv 1 - \mathcal{P}^{\delta}_{\mathcal{D}}(\{\pi \in paths(\mathcal{D}) \mid trace(\pi) \notin traces(\mathcal{L})\}) & \text{– (1)} \\
\equiv 1 - \mathcal{P}^{\delta}_{\mathcal{D}}(G) & \text{– (2)}
\end{array}$$

(1) is true because for every path of $\mathcal{M}$, there is a path of $\mathcal{D}$ with the same probability (as $\mathcal{L}$ is non-probabilistic) and the same trace; and vice versa. (2) is true because by [26], a path of $\mathcal{D}$ such that $trace(\pi) \notin traces(\mathcal{L})$ if and only if it visits some state in G. It can be shown then $\mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{L})$, which is $\mathcal{P}^{max}_{M}(\{\pi \in paths(\mathcal{M}) \mid trace(\pi) \in traces(\mathcal{L})\})$, is $1 - \mathcal{P}^{min}_{\mathcal{D}}(G)$ and $\mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{L})$ is $1 - \mathcal{P}^{max}_{\mathcal{D}}(G)$. □

Intuitively, the theorem holds because, with any scheduler, the probability of $\mathcal{M}$ *not* refining $\mathcal{L}$ is exactly the probability of reaching G in $\mathcal{M} \times nl(\mathcal{L})$. As a result, refinement checking is reduced to reachability probability in $\mathcal{D}$. There are known approaches to compute $\mathcal{P}^{max}_{\mathcal{M}}(G)$ and $\mathcal{P}^{min}_{\mathcal{M}}(G)$, e.g., using an iterative approximation method or by solving linear programs [4].

4.2 SE-LTL Probabilistic Model Checking as Refinement Checking

Another way of specifying properties is through temporal logic. In this section, we examine the problem of model checking PCSP# models against SE-LTL formulae. SE-LTL is an effective property language for PCSP# as it can be constituted by state propositions as well as events. In the pacemaker example, an SE-LTL formula could be stated as follows: $(\Box count \leq 10) \wedge \Box(missingPulseA \Rightarrow X\ pulseB)$ which states $count$ must be always less than 10 and event $missingPulseA$ must lead to an occurrence of event $pulseB$ next. Given an MDP $\mathcal{M}$ and an SE-LTL formula ϕ, let $\mathcal{P}^{max}_{\mathcal{M}}(\phi)$ (and $\mathcal{P}^{min}_{\mathcal{M}}(\phi)$) denote the maximum (and minimum) probability of $\mathcal{M}$ satisfying ϕ.

A standard LTL probabilistic model checking method is the automata-theoretic approach [4]. Firstly, a deterministic Rabin automaton, which is equivalent to the property, is built. The product of the automaton and the system model is then computed.

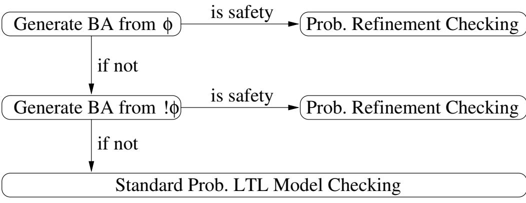


Fig. 2. Workflow

Thirdly, *end components* (which is similar to strongly connected components) in the product which satisfy the Rabin acceptance condition are identified. Lastly, the probability of reaching any state of the end components are calculated, which is exactly the probability of the model satisfying the property. This method is computationally expensive due to multiple reasons. Firstly, the construction of the deterministic Rabin automaton is expensive. Given a Büchi automaton $\mathcal{B}$, its equivalent deterministic Rabin automaton, in the worse case, is of size $2^{\mathcal{O}(n \log n)}$ where n is the size of $\mathcal{B}$. Secondly, identifying the end components is expensive. The worse case complexity is bounded by $\#S \times (\#S + \#T)$ where $\#S$ is the number system states and $\#T$ is the number of the system transitions. In this section, we show that by recognizing safety properties, we can improve probabilistic model checking of certain class of SE-LTL formulae by avoiding constructing the Rabin automaton or computing the end components.

Given a formula ϕ, we check whether ϕ is a safety property using the following approach. Firstly, we check whether it is a *syntactic LTL safety formula* [28]. If it is not, we generate an equivalent Büchi automaton using an existing approach [11], and then check whether all states of the Büchi automaton are accepting. If positive, by the result proved in [1], ϕ is a safety property. If we cannot conclude that ϕ is safety, we assume that it is not. This is a sound but not complete method for recognizing safety. In practice, we found that it is effective in recognizing most of the commonly used safety properties, including for example $\Box(a \Rightarrow \Box b)$ and $\Diamond a \Rightarrow \Box b$.

Next, we adopt the workflow shown in Figure 2 to improve probabilistic model checking. Let ϕ be an SE-LTL formula and $\mathcal{B}$ be the equivalent Büchi automaton. If ϕ is a safety property, then $\mathcal{B}$ can be simply treated as an LTS, as discussed in Section 2. The problem of model checking a system model $\mathcal{M}$ against ϕ is thus reduced to calculate the probability of $\mathcal{M}$ refines the LTS $\mathcal{B}$. If ϕ cannot be determined as a safety property, then we check whether ϕ is a co-safety property. A Büchi automaton $\mathcal{B}'$, equivalent to $\neg\phi$, is generated. If $\mathcal{B}'$ is a safety property, the problem of model checking ϕ is thus reduced to calculate the probability of $\mathcal{M}$ refines the LTS $\mathcal{B}'$.

Theorem 2. *Let $\mathcal{M}$ be an MDP; ϕ be an SE-LTL formula; $\mathcal{B}$ be the Büchi automaton equivalent to ϕ. Let $\mathcal{B}^{-1}$ be the Büchi automaton equivalent to $\neg\phi$. If ϕ is safety, then $\mathcal{P}^{max}_{\mathcal{M}}(\phi) = \mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{B})$ and $\mathcal{P}^{min}_{\mathcal{M}}(\phi) = \mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{B})$; If ϕ is co-safety, then $\mathcal{P}^{max}_{\mathcal{M}}(\phi) = 1 - \mathcal{P}^{min}(\mathcal{M} \sqsupseteq \mathcal{B}^{-1})$ and $\mathcal{P}^{min}_{\mathcal{M}}(\phi) = 1 - \mathcal{P}^{max}(\mathcal{M} \sqsupseteq \mathcal{B}^{-1})$.* □

The proof of the theorem is sketched as follows. If ϕ is a safety property, any trace of $\mathcal{B}$ is accepting. It can be shown that any trace of $\mathcal{M}$ which is not a trace of $\mathcal{B}$ is a

counterexample to ϕ. Therefore, the probability of $\mathcal{M}$ exhibiting a trace of $\mathcal{B}$ (i.e., the probability of $\mathcal{M}$ trace-refines $\mathcal{B}$) is the probability of $\mathcal{M}$ satisfying ϕ. Next, the theorem states that the probability of $\mathcal{M}$ not-refining $\mathcal{B}$ is the probability of $\mathcal{M}$ executing a finite prefix of any traces which is not possible for $\mathcal{B}$. Similarly, we can prove the result for co-safety properties.

By the theorem, probabilistic model checking of safety LTL formula or co-safety LTL formula is reduced to probabilistic refinement checking, which is considerably more efficient as we avoid constructing the deterministic Rabin automaton or identifying end components. This is confirmed by the experiments conducted in Section 5.

5 Case Studies

Our methods have been implemented in the PAT[3] model checker [31]. PAT is a self-contained framework for system modeling, simulation and verification. It supports a layered system design so that new modeling languages and new model checking algorithms/techniques can be easily incorporated. In this paper, we extend PAT with a module to support PCSP#, integrating the existing CSP# language with probabilistic choices. Furthermore, we extend the library of model checking algorithms in PAT with probabilistic refinement checking and probabilistic SE-LTL model checking with safety recognition. We evaluate our implementation using benchmark systems. We compare our results with PRISM version 3.3.1. In order to perform a fair comparison, we use existing PRISM models; re-model them using the extended CSP# language and re-verify them using PAT. It should be noticed that our language is capable of specifying hierarchical systems which are beyond PRISM. Working with existing PRISM models, which are not hierarchical, is not justified to show our advantage. Nonetheless, we show that even for those systems, PCSP# offers an intuitive and compact representation and PAT offers comparable performance. The following models are adopted for comparison.

- Model ME describes a probabilistic solution to *N*-process mutual exclusion problem, which is based on [25].
- Model RC is a shared coin protocol of the randomized consensus algorithm, which is based on [3]. Note that *N* is the number of coins and *K* is a parameter used to generate suitable probability.
- Model DP is the probabilistic *N*-dining philosophers under fairness, based on [19].
- Model CS is the IEEE 802.3 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) protocol, which is based on [23]. Note that *N* is the number of stations and *K* is the exponential backoff limit.

The models (and others) with configurable parameters are embedded in the latest version of PAT. In the following, we discuss three aspects of the comparison.

Comparison on modeling. The simplicity of writing models is not purely a matter of aesthetics. It may yield gains in time and space. Table 1 presents the size of the models (in number lines of codes) as well as the number of global states. The size of all models are reduced. Note that with different parameters, the PRISM models vary in sizes,

[3] Available at http://pat.comp.nus.edu.sg

Table 1. Experiments on modeling

System	PAT			PRISM		
	LOC	#States	Deadlock Check(s)	LOC	#States	Deadlock Check(s)
ME (N=5)	22	5489	0.351	36	308800	0.410
ME (N=8)	22	86966	7.279	39	390068480	1.203
RC (N=4, K=4)	24	6835	0.218	25	43136	0.110
RC (N=10, K=6)	24	997403	56.072	31	7598460928	3.250
DP (N=5)	20	32766	2.413	30	93068	0.156
DP (N=6)	20	260100	25.775	31	917424	0.672
CS (N=2, K=4)	115	9165	0.337	119	7958	0.266
CS (N=3, K=2)	94	49101	2.243	122	36850	0.772

whereas the size of the PAT models remain constant. The state counts for PAT models are significantly smaller than those of the PRIMS models. The state counts are reported by PAT and PRISM when checking deadlock-freeness of both models. One of the reasons why PAT may have much less states is that global variables in the PRISM models, which are used to track local state of each processes, are removed (and become part of the process definition). The processes then become fully symmetric (as expected in the original protocol), which then triggered an internal state reduction based on symmetry reduction in PAT.

Performance of refinement checking. In general, refinement checking and temporal logic verification are good at different classes of properties. For instance, using temporal logic formulae to capture the process $OKHrt$ (shown in Section 4) would result in a large formula which in turn result in in-efficient verification. Our experiments, however, show that even for those properties designed for temporal-logic based verification, probabilistic refinement checking offers comparable performance. Given any safety property of the above mentioned models, we build a property model and verify the property by refinement checking. Table 2 presents the experiment results. The experiment data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 2GB RAM. We use the iterative method in calculating the probability and set termination threshold as relative difference 1.0E-6 (exactly same as PRISM). PAT performs worse than PRISM for ME, comparable for RC and better for DP. The main reason that PAT outperforms PRISM for the DP model is that PAT has less states and its refinement checking algorithm has less computation than temporal logic-based model checking. Note that because the models are designed to satisfy the properties, the result probability is all 1.

Performance improvement using safety recognition. Lastly, we show that safety recognition improves probabilistic LTL model checking and allows PAT to outperform PRISM in many cases. Safety recognition in PAT is based on syntax analysis or simple heuristics based on the generated Büchi automata. The computational overhead is negligible. Table 3 presents the experiment results on verifying the models against safety, co-safety and properties which are neither. Column $PAT(w)$ ($PAT(w/o)$) shows the time taken with (without) safety recognition. If the property is neither safety or co-safety, safety recognition becomes computational overhead. The cost is however negligible as evidenced in the table. For safety or co-safety properties, PAT performs better

Table 2. Experiments on refinement checking

System	Property	Result(Pmax)	PAT (s)	PRISM (s)
ME (N=5)	mutual exclusion	1	0.359	0.282
ME (N=8)	mutual exclusion	1	9.831	1.234
ME (N=10)	mutual exclusion	1	81.192	3.127
RC (N=4,K=4)	consensus	1	0.218	0.328
RC (N=6,K=6)	consensus	1	2.813	2.543
RC (N=8,K=8)	consensus	1	19.642	14.584
DP (N=5)	once eat, never hungry	1	3.333	37.769
DP (N=6)	once eat, never hungry	1	53.062	389.334

Table 3. Experiments on LTL checking

System	Property	Result(Pmax)	PAT (w)	PRISM	PAT (w/o)
ME (N=5)	co-safety	1	2.356	231.189	27.411
ME (N=8)	co-safety	1	94.204	-	8901.295
ME (N=10)	co-safety	1	1076.217	-	-
RC (N=4,K=4)	co-safety(1)	0.99935	0.379	21.954	12.150
RC (N=4,K=4)	neither	0.54282	6.106	45.612	6.087
RC (N=4,K=4)	co-safety(2)	0.15604	6.703	35.144	7.868
RC (N=6,K=6)	co-safety(1)	1	5.854	1755.984	585.706
RC (N=6,K=6)	neither	0.53228	457.815	-	442.008
RC (N=6,K=6)	co-safety(2)	0.12493	355.027	-	453.362
RC (N=8,K=8)	co-safety(1)	1	52.906	-	-
RC (N=8,K=8)	neither	0.52537	10179.796	-	10107.268
RC (N=8,K=8)	co-safety(2)	0.10138	5923.086	-	9420.430
DP (N=5)	safety	1	1.162	37.769	10.006
DP (N=6)	safety	1	9.760	389.334	164.423
DP (N=5)	co-safety	1	1.039	38.347	544.307
DP (N=6)	co-safety	1	9.091	384.231	-
CS (N=2, K=4)	co-safety(1)	1	0.615	0.921	0.736
CS (N=2, K=4)	co-safety(2)	0.99902	0.933	2.314	1.034
CS (N=3, K=2)	co-safety(1)	1	6.118	1.733	6.707
CS (N=3, K=2)	co-safety(2)	0.85962	6.284	7.233	7.484

with safety recognition. In comparison with PRISM, PAT outperforms PRISM (for almost all properties) for some models, e.g., ME and RC. This is mainly because the PAT models have much less states, because of the difference in modeling. For some other models (e.g., DP and CS), safety recognition allows PAT to outperform PRISM.

In general, PRISM handles more states per time unit than PAT. This is suggested by the experiment results presented in Table 1, which shows the time for verifying deadlock-freeness. Apart from the fact that PRISM has been optimized for many years, the main reason is the complexity in handling hierarchical models. Note that though these models have simple structures, there is overhead for maintaining underlying data structures designed for hierarchical systems. PRISM is based on MTBDD, whereas PAT is based on explicit state representation currently. Symbolic methods like BDD are

known to handle more states [7]. Applying BDD techniques to hierarchical complex languages like PCSP# is highly non-trivial. It remains as one of our ongoing work. The experiment results are not to be taken as the limit of PAT. The fact that PAT handles less states per time unit does not imply that PAT is always slower than PRISM, as evidenced in the experiments. The main reason is that 1) a system modeled using PRISM may have more states than its model in PCSP# due to its language limitation; 2) safety/co-safety recognition which avoid much computation in probabilistic model checking.

6 Conclusion

The main contribution of this work is the extended PAT model checker which offers a self-contained framework for modeling and checking of hierarchical complex probabilistic systems. Compared to existing probabilistic model checkers, PAT offers an expressive modeling language and an alternative way of probabilistic system verification, i.e., refinement checking. In addition, PAT improves LTL probabilistic model checking by supporting SE-LTL and safety recognition.

As for future research directions, we will explore methods for checking refinement relationship between probabilistic PCSP# models. Furthermore, We are investigating how to combine zone abstraction for real-time systems with probabilistic system behaviors so that we can support real-time probabilistic systems. In addition, in order to tackle the state space explosion problem, optimization techniques like partial order reduction and symmetry reduction will be incorporated.

References

1. Alpern, B., Schneider, F.B.: Recognizing Safety and Liveness. Distributed Computing 2(3), 117–126 (1987)
2. Alur, R., Henzinger, T.A.: Reactive Modules. Formal Methods in System Design 15(1), 7–48 (1999)
3. Aspnes, J., Herlihy, M.: Fast Randomized Consensus Using Shared Memory. Journal of Algorithms 15(1), 441–460 (1990)
4. Baier, C., Katoen, J.: Principles of Model Checking. The MIT Press, Cambridge (2008)
5. Barold, S.S., Stroopbandt, R.X., Sinnaeve, A.F.: Cardiac Pacemakers Step by Step: an Illustrated Guide. Blackwell Publishing, Malden (2004)
6. Bellman, R.: A Markovian Decision Process. Journal of Mathematics of Mechanics 6 (1957)
7. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic Model Checking: 10^{20} States and Beyond. Inf. Comput. 98(2), 142–170 (1992)
8. 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)
9. Chen, Y., Sanders, J.W.: Unifying Probability with Nondeterminism. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 467–482. Springer, Heidelberg (2009)
10. Ciesinski, F., Baier, C.: LiQuor: A Tool for Qualitative and Quantitative Linear Time Analysis of Reactive Systems. In: QEST, pp. 131–132. IEEE Computer Society, Los Alamitos (2006)
11. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
12. Geilen, M.: On the Construction of Monitors for Temporal Logic Properties. Electr. Notes Theor. Comput. Sci. 55(2) (2001)

13. Havelund, K., Rosu, G.: Synthesizing Monitors for Safety Properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
14. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A Tool for Automatic Verification of Probabilistic Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
15. Hoare, C.: Communicating Sequential Processes. International Series in Computer Science. Prentice-Hall, Englewood Cliffs (1985)
16. Katoen, J., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The Ins and Outs of the Probabilistic Model Checker MRMC. In: QEST, pp. 167–176. IEEE Computer Society, Los Alamitos (2009)
17. Kupferman, O., Vardi, M.Y.: Model Checking of Safety Properties. Formal Methods in System Design 19(3), 291–314 (2001)
18. Latvala, T.: Efficient Model Checking of Safety Properties. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 74–88. Springer, Heidelberg (2003)
19. Lehmann, D., Rabin, M.: On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract). In: POPL, pp. 133–138. ACM, New York (1981)
20. Maisel, W.H., Moynahan, M., Zuckerman, B.D., Gross, T.P., Tovar, O.H., Tillman, D., Schultz, D.B.: Pacemaker and ICD Generator Malfunctions. The Journal of American Medical Association 295(16), 1901–1906 (2006)
21. Morgan, C., Hoang, T.S., Abrial, J.: The Challenge of Probabilistic *Event B* - Extended Abstract. In: Treharne, H., King, S., Henson, M.C., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 162–171. Springer, Heidelberg (2005)
22. Morgan, C., McIver, A., Seidel, K., Sanders, J.W.: Refinement-Oriented Probability for CSP. Formal Asp. Comput. 8(6), 617–647 (1996)
23. Nicollin, X., Sifakis, J., Yovine, S.: Compiling Real-time Specifications into Extended Automata. IEEE Transactions on Software Engineering 18(9), 794–804 (1992)
24. Pnueli, A.: The Temporal Logic of Programs. In: FOCS, pp. 46–57. IEEE, Los Alamitos (1977)
25. Pnueli, A., Zuck, L.: Verification of Multiprocess Probabilistic Protocols. Distributed Computing 1(1), 53–72 (1986)
26. Roscoe, A.W.: Model-checking CSP, pp. 353–378 (1994)
27. 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)
28. Sistla, A.P.: Safety, Liveness and Fairness in Temporal Logic. Formal Asp. Comput. 6(5), 495–512 (1994)
29. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
30. Sun, J., Liu, Y., Dong, J.S., Chen, C.Q.: Integrating Specification and Programs for System Modeling and Verification. In: TASE, pp. 127–135. IEEE Computer Society, Los Alamitos (2009)
31. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
32. Vardi, M.Y., Wolper, P.: An Automata-Theoretic Approach to Automatic Program Verification. In: LICS, pp. 332–344. IEEE Computer Society, Los Alamitos (1986)
33. Zhu, H., Qin, S., He, J., Bowen, J.: PTSC: Probability, Time and Shared-Variable Concurrency. International Journal on Innovations in Systems and Software Engineering 5(4), 271–294 (2009)

Trace-Driven Verification of Multithreaded Programs*

Zijiang Yang[1] and Karem Sakallah[2]

[1] Western Michigan University, Kalamazoo, MI 49008, USA
[2] University of Michigan, Ann Arbor, MI 48109, USA

Abstract. We present a new method that combines the efficiency of testing with the reasoning power of satisfiability modulo theory (SMT) solvers for the verification of multithreaded programs under a user specified test vector. Our method performs dynamic executions to obtain both under- and over-approximations of the program, represented as quantifier-free first order logic formulas. The formulas are then analyzed by an SMT solver which implicitly considers all possible thread interleavings. The symbolic analysis may return the following results: (1) it reports a real bug, (2) it proves that the program has no bug under the given input, or (3) it remains inconclusive because the analysis is based on abstractions. In the last case, we present a refinement procedure that uses symbolic analysis to guide further executions.

1 Introduction

One of the main challenges in testing multithreaded programs is that the absence of bugs in a particular execution does not necessarily imply error-free operation under that input. To completely verify program behavior for a given test input, *all executions* permissible under that input must be examined. However, this is often an infeasible task considering the exponentially large number of possible interleavings of a typical multithreaded program. A program with n threads, each executing k statements, can have up to $(nk)!/(k!)^n \geq (n!)^k$ thread interleavings, a dependence that is exponential in both n and k.

In this paper we address this challenge by an approach called *Trace-Driven Verification* (TDV) that combines the efficiency of testing with the reasoning power of satisfiability modulo theory (SMT) solvers. TDV performs dynamic executions to obtain approximations, represented as quantifier-free first order logic (FOL) formulas, of the program under verification. The formulas are then analyzed by an SMT solver which implicitly considers all possible thread interleavings. The symbolic analysis may return one of the following results: (1) it reports a real bug, (2) it proves that the program has no bug under the given input, or (3) it remains inconclusive because the analysis is based on abstractions. In the last case, we present a refinement procedure that uses symbolic analysis to guide further executions. The features of TDV include:

* The work was supported by NSF Grant CCF-0811287.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 404–419, 2010.
© Springer-Verlag Berlin Heidelberg 2010

- **Implicit consideration of thread interleavings.** As explicit enumeration of executions is intractable, the alternative we present is to capture thread interleavings implicitly as a set of constraints in a satisfiability formula. These constraints belong to the family of quantifier-free first order logic formulas for which efficient SMT solvers are available.
- **Integration of dynamic executions and symbolic analysis.** At any given time, TDV analyzes only the statements that appear in a particular execution under a user-specified test vector. It may report a real bug, or prove that the program behaves as expected under all thread interleavings stimulated by the given input. In either case, TDV avoids the analysis of statements that do not appear in an execution. However, it is also possible that the symbolic analysis, being an abstraction of program behavior, remains inconclusive. In such a case, TDV uses the symbolic analysis result to guide future concrete executions.
- **Abstraction with both under- and over-approximations.** Based on an execution, TDV infers both under- and over-approximations of the entire program. The under-approximation is complete so that any bug detected in the model is a real bug; while the over-approximation is sound so that it can be used to prove the absence of bugs.

The rest of the paper is organized as follows. After giving the algorithm overview in Section 2, we present the symbolic encoding of program traces in Section 3. The refinement procedure is illustrated in Section 4. In Section 5 we outline several encoding and algorithmic optimizations to improve scalability. We discuss related work in Section 6. Finally we present experimental results in Section 7 and conclude the paper in Section 8.

2 Algorithm Overview

Consider a multithreaded program P where threads communicate via shared variables. Without loss of generality, we assume there is at most one shared variable access at a program statement[1]. Then each statement constitutes an *atomic computational step*, at which granularity the thread scheduler can switch control between threads during the execution.

Consider the program, shown in Fig. 1, that consists of two concurrently running threads. In a typical testing environment, even if we run the program multiple times under the test input $a = 1, b = 0$, we may obtain the same executed trace $\pi_1 = \langle 1, 2, 5, 6, 7 \rangle$ where the integer values indicate the line numbers. In general, an executed trace is an ordered sequence of program statements executed by the different threads. Although π_1 does not cause an assertion failure

[1] If there are multiple shared variable accesses in one statement, we can introduce additional local variables and split the statement into multiple statements such that each statement has at most one shared variable. For example, consider a statement $a = x + y$ with shared variables x, y and local variable a. It can be split into two statements $t = y$ and $a = x + t$ with the help of a temporary local variable t.

```
Thread 1:                          Thread 2:

foo (int a) {                      bar (int b) {
1    y = a + 1;                    6    if (b >= 0)
2    if (y < 2)                    7        y = b + 1;
3        complexA();               8    else
4    else                          9        complexB();
5        assert(y >= 2);           }
}
```

Fig. 1. A program with the shared variable y and local variables a, b

on Line 5, we cannot conclude the absence of assertion failures in this program as this input admits other interleavings of these two threads. Table 1 shows the set $\Pi(\pi_1)$ of all 10 possible interleavings of π_1. For each trace in the table, the bottom row indicates whether the assertion on Line 5 holds (h) or fails (f). However, not all the interleavings in $\Pi(\pi_1)$ are valid executions. Closer examination of π_6 and π_9 shows that they are infeasible traces, due to the violation of program semantics. In particular, after y is updated by Thread 2 on Line 7, it is not possible for Thread 1 to follow the `Else` branch on Line 2. Let $\Pi_P(\pi_1)$ be the set of interleavings derived from π_1 that are consistent with the semantics of the program P. We have $\Pi_P(\pi_1) = \{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_7, \pi_8, \pi_{10}\}$. We call a trace $\pi_i \in \Pi_P(\pi_1) \backslash \{\pi_1\}$ an *induced trace* of π_1.

Table 1. $\Pi(\pi_1)$: all the thread interleavings of π_1. The two interleavings marked with an asterisk are invalid since they violate program semantics.

Step	π_1	π_2	π_3	π_4	π_5	π_6^*	π_7	π_8	π_9^*	π_{10}
1	1	1	1	1	1	1	6	6	6	6
2	2	2	2	6	6	6	1	1	1	7
3	5	6	6	2	2	7	2	2	7	1
4	6	5	7	5	7	2	5	7	2	2
5	7	7	5	7	5	5	7	5	5	5
assert	h	h	f	h	f	f	h	f	f	h

In order to check for assertion failures not only in π_1 but also in its induced traces, we construct an FOL formula $\varphi(\pi)$ that implicitly models all the traces in $\Pi_p(\pi)$ (see Section 3.1 for details). A satisfying assignment to $\varphi(\pi)$ indicates a true assertion failure and can be used to identify the particular thread interleaving that produces it. If $\varphi(\pi)$ is unsatisfiable, however, we cannot conclude correctness because $\varphi(\pi)$ is an under-approximation of program behavior. To understand the reason consider a statement `assert(`$C_{complexA}$`)` inside `complexA()` on Line 3 in Fig. 1. Given the executed trace $\pi_1 = \langle 1, 2, 5, 6, 7 \rangle$, $\varphi(\pi_1)$ itself cannot reveal any assertion failure inside `complexA()` since the `assert(`$C_{complexA}$`)` statement does not even appear in any traces of $\Pi_p(\pi_1)$. On the other hand, there exist valid executions that execute `complexA()` (e.g. $\pi' = \langle 1, 6, 7, 2, 3, \ldots \rangle$). Thus an assertion failure is still possible under the test input $a = 1, b = 0$.

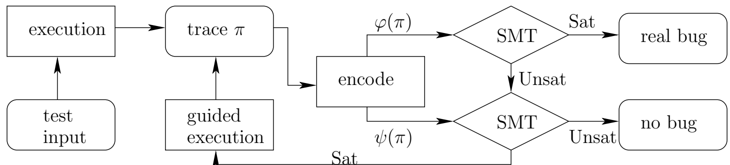


Fig. 2. Trace-driven verification flow

To insure correctness (absence of assertion failures), all execution traces permissible under that input must be examined. We relax, or abstract $\varphi(\pi)$, by making changes to and dropping some of its constraints (see Section 3.2 for details). This leads to $\psi(\pi)$, an FOL formula that represents an over-approximation to the program behavior under the specified input. If $\psi(\pi)$ is unsatisfiable, we can provably conclude the absence of assertion failures for all thread interleavings under the specified input. Otherwise we need to check if the reported violation is true or spurious. In the latter case, TDV performs refinement by modifying the control flow in order to examine other executions of P under the same test input.

As illustrated in Fig. 2, TDV consists of the following steps:

1. Run the program under a given user input to obtain an initial execution trace π.
2. Using an encoding along the lines illustrated in Section 3.1, construct an FOL formula $\varphi(\pi)$.
3. Using an SMT solver, check the satisfiability of $\varphi(\pi)$.
   - If $\varphi(\pi)$ is found to be satisfiable, a real bug is found. Based on the solution to $\varphi(\pi)$ we can report to the user the specific scheduling that exposes the bug.
   - If $\varphi(\pi)$ is found to be unsatisfiable, we relax $\varphi(\pi)$ to obtain $\psi(\pi)$. This allows us to examine *sibling* traces, i.e., traces that conform to the same input but cover different statements.
     - If $\psi(\pi)$ is found to be unsatisfiable, we can conclude that the property holds under all possible thread interleavings under the given test input.
     - If $\psi(\pi)$ is found to be satisfiable, the SMT solver returns a counter-example, which is used to guide new executions that are guaranteed to touch new statements that have not appeared in previous executions.

3 Symbolic Encoding of Execution Traces

An executed trace is a sequence $\pi = \langle (t_1, l_1.o_1, Q_1), \ldots, (t_n, l_n.o_n, Q_n) \rangle$ that lists the statements executed by the various threads. Each tuple $(t, l.o, Q) \in \pi$ is considered to be an atomic computational step where t is the thread id, l is the line number for the statement, o is an *occurrence index* that distinguishes

the different executions of the same statement, and Q is the statement type that can be one of *assign, branch, jump, fork, join* or *assert*. In this paper we assume all the executions eventually terminate[2].

We consider three basic types of statements: assignment $v = E$ where E is an arithmetic expression, branch $C?l.o$ where C is a relational expression, and jump *goto* $l.o$. Note that $C?l.o$ only lists the destination if C holds because no two branches can be taken simultaneously in an executed trace[3]. Besides the basic types, we also allow `assert(C)` for checking assertions, `exit` for signaling the termination of a thread, and the synchronization primitives. `fork(t)` and `join(t)` allow a thread to dispatch and wait for the completion of another Thread t. Given a program written in a full-fledged programming languages like C, one can use pre-processing [21] to simplify its executed traces into the basic statements described above.

3.1 Under-Approximation FOL Formula $\varphi(\pi)$

The key to the TDV algorithm is the construction of appropriate FOL formulas that can be easily checked with SMT solvers.

Let V_G and $V_L(t)$ denote the set of global and local variables in Thread t, respectively. Let the set of variables visible to t be $V(t) = V_G \cup V_L(t)$. In addition to program variables, we introduce a statement location variable L_t for each thread, whose domain includes all the possible line numbers and occurrence indices. To model nondeterminism in the scheduler, we add a variable T whose domain is the set of thread indices. A transition in Thread t is executed only when $T = t$. At every transition step we add a fresh copy for each variable. That is, $v[i]$ denotes the copy of v at the i-th step. Given an executed trace π, $\varphi(\pi)$ consists of following constraints:

– **Program transition constraint** δ_π that expresses the effect of executing a particular statement of the program by a particular thread. For each tuple $(t, l.o, Q)$ except when Q is *exit*, we assume the next tuple to be executed by Thread t is $(t, l'.o', Q')$. Once the last tuple $(t, l.o, exit)$ of Thread t has been executed, we use $\triangle$ to indicate the end of Thread t. Let $\delta_{t,l.o}[i]$ denote the constraints of $(t, l.o, Q) \in \pi$ at step i. Fig. 3 shows the encoding for different types of tuples. For example, the one for $(t, l.o, v = E)$ states that if Thread t executes the statement at step i, the following updates occur at step $i + 1$:
  1. the next statement for Thread t to execute is $l'.o'$;
  2. the value of v at step $i + 1$ is $E|_{V \to V[i]}$ with all variables in E replaced by their corresponding versions at step i; and
  3. other visible variables remain unchanged.

[2] For nonterminating programs, our procedure can be used as a bounded analysis tool to search for bugs up to a bounded number of execution steps.

[3] A conditional branch such as `if C then` $l_1 : \ldots$ `else` $l_2 : \ldots$ results in the executed trace $C?l_1$ if the `then` branch is executed, and $\neg C?l_2$ otherwise.

The program transition constraint δ_π is defined as

$$\delta_\pi \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{(t,l.o)} \delta_{t,l.o}[i] \quad (1)$$

assignment: (t, l.o, v = E)

$T[i] = t \wedge L_t[i] = l.o \rightarrow$

$L_t[i+1] = l'.o' \wedge v[i+1] = E|_{V \rightarrow V[i]} \wedge V(t)\backslash v[i+1] = V(t)\backslash v[i]$

conditional branch: (t, l.o, C?l'.o')

$T[i] = t \wedge L_t[i] = l.o \wedge C|_{V \rightarrow V[i]} \rightarrow L_t[i+1] = l'.o' \wedge V(t)[i+1] = V(t)[i]$

thread termination: (t, l.o, exit)

$T[i] = t \wedge L_t[i] = l.o \rightarrow L_t[i+1] = \Delta \wedge V(t)[i+1] = V(t)[i]$

unconditional jump: (t, l.o, goto l'.o')

$T[i] = t \wedge L_t[i] = l.o \rightarrow L_t[i+1] = l'.o' \wedge V(t)[i+1] = V(t)[i]$

thread fork: (t, l.o, fork(t'))

$T[i] = t \wedge L_t[i] = l.o \rightarrow$

$L_t[i+1] = l'.o' \wedge L_{t'}[i+1] = s_{t'} \wedge V(t)[i+1] = V(t)[i]$

thread join: (t, l.o, join(t'))

$$\begin{pmatrix} T[i] = t \wedge L_t[i] = l.o \wedge L_{t'}[i] = \Delta \rightarrow \\ L_t[i+1] = l'.o' \wedge V(t)[i+1] = V(t)[i] \end{pmatrix} \wedge$$

$$\begin{pmatrix} T[i] = t \wedge L_t[i] = l.o \wedge L_{t'}[i] \neq \Delta \rightarrow \\ L_t[i+1] = l.o \wedge V(t)[i+1] = V(t)[i] \end{pmatrix}$$

lock: (t, l.o, lock(lk))

$$\begin{pmatrix} T[i] = t \wedge L_t[i] = l.o \wedge \neg lk[i] \rightarrow \\ L_t[i+1] = l'.o' \wedge lk[i+1] = true \wedge V(t)\backslash lk[i+1] = V(t)\backslash lk[i] \end{pmatrix} \wedge$$

$(T[i] = t \wedge L_t[i] = l.o \wedge lk[i] \rightarrow L_t[i+1] = l.o \wedge V(t)[i+1] = V(t)[i])$

unlock: (t, l.o, unlock(lk))

$T[i] = t \wedge L_t[i] = l.o \rightarrow$

$L_t[i+1] = l'.o' \wedge lk[i+1] = false \wedge V(t)\backslash lk[i+1] = V(t)\backslash lk[i]$

Fig. 3. Program transition constraints. $T[i]$ is the active thread at step i; $L_t[i]$ ($L_t[i+1]$) is the statement location at step i ($i+1$); $E|_{V \rightarrow V[i]}$($C|_{V \rightarrow V[i]}$) substitute all variables in $E(C)$ by the by their corresponding versions at step i; $V(t)\backslash v[i+1] = V(t)\backslash v[i]$ denotes all visible variables in t keep their value except variable v.

– **Initial condition constraint** ι_π that specifies the starting locations for each thread as well the initial values of program variables, including the values set by the input vector.

– **Trace enforcement constraint** ε_π that restricts the encoded behavior to include only the statements appearing in an executed trace π. For each $(t, l.o, C?l'.o') \in \pi$ we assume condition C holds on line l at o-th occurrence in π. Then we have

$$\varepsilon_\pi \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{(t,l.o)} (T[i] = t \wedge L[i] = l.o \rightarrow C|_{V \rightarrow V[i]}) \qquad (2)$$

– **Thread control constraint** τ_π that (1) insures that the local state of a thread (the values of its local variables) remains unchanged when the thread is not executing, and (2) insures that the thread cannot be selected for execution after it has terminated. These two constraints are specified in Equation 3.

$$\begin{aligned} \tau_{t,idle}[i] &\equiv T[i] \neq t \rightarrow L_t[i+1] = L_t[i] \wedge V_L(t)[i+1] = V_L(t)[i] \\ \tau_{t,done}[i] &\equiv L_t[i] = \Delta \rightarrow T[i] \neq t \end{aligned} \qquad (3)$$

The thread control constraint is defined as follows:

$$\tau_\pi \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{t=1}^{N} (\tau_{t,idle}[i] \wedge \tau_{t,done}[i] \wedge \tau_{other}) \qquad (4)$$

In τ_{other}, additional optional constraints can be included to model particular scheduling policy.

– **Property constraint** ρ_P that indicates the correctness conditions, specified as assertions within the program in this paper, that we would like to check for validity under all possible executions. Note that many common programming errors can be modeled as assertions [21]. Let $(t, l, assert(C))$ be an assertion on line l in Thread t. The property constraint can be specified as follows:

$$\rho_P \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{(t,l)} (T[i] = t \wedge L[i] = l \rightarrow C|_{V_C \rightarrow V_C[i]}) \qquad (5)$$

Note that properties encoded by ρ_P are not necessarily the assertions appearing in π only; the assertions may appear anywhere in the program P. This is a crucial requirement for our trace-based method to find real failures anywhere in the program, or to prove the absence of assertion failures of the program.

Whether the property ρ_P holds for all possible thread interleavings in $\Pi_P(\pi)$ is determined by checking the validity of the formula: $\iota_\pi \wedge \delta_\pi \wedge \tau_\pi \wedge \varepsilon_\pi \rightarrow \rho_P$, which is equivalent to checking the *satisfiability* of the formula

$$\varphi(\pi) \equiv \iota_\pi \wedge \delta_\pi \wedge \tau_\pi \wedge \varepsilon_\pi \wedge \neg\rho_P \qquad (6)$$

Equation 6, which implicitly represents all thread interleavings of $\Pi_P(\pi)$, is still an under-approximation of the behavior of program P under the given test input. Therefore, a solution to $\varphi(\pi)$ reveals real errors in the program, but the unsatisfiability of $\varphi(\pi)$ does not prove the absence of errors.

3.2 Over-Approximation FOL Formula $\psi(\pi)$

Let $\Pi_P(\overrightarrow{v})$ be the set of all possible execution traces of program P under the test input $\overrightarrow{v}$. The set of interleavings considered by $\varphi(\pi)$ is $\Pi_P(\pi) \subseteq \Pi_P(\overrightarrow{v})$.

To catch assertion violations in branches not yet executed in π, or to establish the absence of such violations in all traces, we need an over-approximation of $\Pi_P(\overrightarrow{v})$. The over-approximated encoding can be obtained from $\varphi(\pi)$ with the following changes:

- Remove the trace enforcement constraint ε_π that prohibits any trace $\pi' \notin \Pi_P(\pi)$ from being considered in $\varphi(\pi)$. In Fig. 1, for example, a trace starting from $\langle 1, 6, 7, 2, 3, \ldots \rangle$ can be a valid execution according to the program. However, the ε_π constraint $T[i] = 1 \wedge L[i] = 2 \rightarrow y[i] \geq 2$ prohibits the trace from being considered.
- Collapse multiple occurrences. For statements that occur more than once, we consider only one instance in the transition constraint. Thus the occurrence index o is no longer needed. This leads to a modified transition constraint δ_π^o.
- Add control flow constraints λ_π for un-executed statements. λ_π keeps the control flow logic but ignores the data logic in those statements that do not occur in π. The purpose of λ_π is to force the over-approximated behavior to at least follow the control flow logic of program P. Here we consider assignments and conditional branches. Given a conditional branch $(t, l, C?l_1 : l_2) \notin \pi$ that executes l_1 next if C is true and l_2 next otherwise, we add a constraint to $\lambda_\pi[i]$:
$$T\left[i\right] = t \wedge L_t\left[i\right] = l \rightarrow L_t\left[i+1\right] = l_1 \vee L_t\left[i+1\right] = l_2. \tag{7}$$
Similarly, for an assignment statement $(t, l, v = E) \notin \pi$ that executes l_1 next, the constraint added to $\lambda_\pi[i]$ is
$$T\left[i\right] = t \wedge L_t\left[i\right] = l \rightarrow L_t\left[i+1\right] = l_1 \tag{8}$$

After the modifications above we obtain the following over-approximation:

$$\psi(\pi) \equiv \iota_\pi \wedge \delta_\pi^o \wedge \tau_\pi \wedge \lambda_\pi \wedge \neg\rho_P \tag{9}$$

Let $\Omega(\pi)$ be the set of interleavings considered by ψ_π; then $\Omega(\pi) \supseteq \Pi_P(\overrightarrow{v})$ is an over-approximation of the program behavior under the test vector $\overrightarrow{v}$. In general, the unsatisfiability of $\psi(\pi)$ proves P has no assertion failures under the test vector $\overrightarrow{v}$. The downside of using $\psi(\pi)$ is the inevitability of invalid executions which need to be filtered out afterwards. In the running example in Fig. 1, the SMT solver may report π_6 in Table 1 as a satisfiable solution of $\psi(\pi)$. However, it is not a feasible trace since the behavior of the step in line 2 is unspecified in $\psi(\pi)$ when $y < 2$.

4 Refinement

4.1 Analysis-Guided Execution

Let CEX_π be a satisfiable assignment to all variables in $\psi(\pi)$; it is called a potential counterexample. In the counterexample guided abstraction refinement

(CEGAR) framework, a decision procedure (theorem prover, SAT solver, or BDDs) [8,2,1,27] has been used to check whether CEX_π is feasible in P, and if not, to refine the over-approximation. Such an approach may not be scalable for handling multithreaded software due to the program complexity and the length of the counterexamples.

Instead, we use guided concrete execution rather than a theorem prover or a SAT solver. Let $T = \cup_{i=1}^{|\pi|}\{T[i]\}$ be the set of thread selection variables at all time steps, and let $L = \cup_{i=1}^{|\pi|} \cup_{t=1}^{N} \{L_t[i]\}$ be the set of line number variables. Given CEX_π, we first extract a thread schedule $SCH_\pi = \exists_{v \in \{T \cup L\}}.CEX_\pi$, and organize it as a sequence

$$\pi_{SCH} = \langle (t_1, l_1), (t_2, l_2), \ldots, (t_{|\pi|}, l_{|\pi|}) \rangle \ .$$

Note that the occurrence index is not needed as the sequence uniquely identifies a trace (although it may be infeasible). The program is then re-executed by trying to follow π_{SCH}; this is implemented by using check-point and restart techniques as in [30]. If the re-execution can follow π_{SCH} to completion, then π_{SCH} represents a real bug. Otherwise, we obtain a new executed trace

$$\pi' = \langle (t_1, l_1.o_1), \ldots, (t_{k-1}, l_{k-1}.o_{k-1}), (t'_k, l'_k.o'_k), \ldots, (t'_{|\pi'|}, l'_{|\pi'|}.o'_{|\pi'|}) \rangle \ .$$

π and π' have the same thread ids and line numbers for the first $k-1$ steps. But starting from the k-th step π' can no longer follow π and completes the execution on its own.

To sum up, by performing a guided execution after analyzing the over-approximation $\psi(\pi)$, we are able to either validate the potential counterexample CEX_π, or obtain a new execution π' for a further analysis.

4.2 Avoid Redundant Checks

To avoid performing symbolic analysis on executed traces that have been analyzed before, we maintain a set χ of already inspected traces. Let $\{\pi_1, \ldots, \pi_m\}$ be the set of executed traces in the first m iterations that have been analyzed. If $\psi(\pi_m)$ is satisfiable, we are only interested in a solution $\overrightarrow{S}$ such that the trace $\pi_{\overrightarrow{S}}$ corresponding to $\overrightarrow{S}$ satisfies $\pi_{\overrightarrow{S}} \notin \Pi_P(\pi_i)$ for all $1 \leq i \leq m$. Such requirement is not only for performance, but also for the termination of the algorithm: without χ our algorithm may analyze the same executed trace infinitely.

Let π_t be a subsequence of π that is executed by Thread t. For two such subsequences π_t^1 and π_t^2 from two different executed traces, if they visit the same set of branch statements in t and have the same truth value of the conditionals at each branch, then $\pi_t^1 \equiv \pi_t^2$ (same statements are visited in the same order). Therefore, the trace enforcement constraint ε_{π_t} uniquely identifies a trace π_t in Thread t. As $\Pi_P(\pi)$ is the interleavings among the traces $\pi_{t_1}, \ldots, \pi_{t_N}$, they are identified by $\varepsilon_\pi = \varepsilon_{\pi_{t_1}} \wedge \ldots \wedge \varepsilon_{\pi_{t_N}}$. In the other words, in order to find a trace

not in $\Pi_P(\pi)$, we must add the constraint $\neg\varepsilon_\pi$. Assume $\{\pi_1, \ldots, \pi_m\}$ are the traces that have been executed so far, we have

$$\chi_m \equiv \bigwedge_{k=1}^{m} \neg\varepsilon_{\pi_k}. \tag{10}$$

The over-approximation formula at the $(m+1)$-th iteration becomes

$$\psi(\pi) \equiv \iota_\pi \wedge \delta_\pi^o \wedge \tau_\pi \wedge \lambda_\pi \wedge \chi_m \wedge \neg\rho_P. \tag{11}$$

4.3 An Illustrative Example

Fig. 4 shows a program with two methods `foo` and `bar`. At Line 0 `foo` creates a new thread and invoke `bar`. There is a recursive call on Line 3 in `foo`, therefore, multiple threads may be created depending on the input value of a. In the program, x and y are global variables with initial value 1, while a and b are thread local variables. We would like to check whether there can be an assertion failure on Line 11 under the test value $a = 1$.

```
foo(int a) {
0   create a new thread t to invoke bar(1);
1   x = a;
2   if (x>0)
3       foo(a-1);
4   else
5       if (y<=1 && y! =0)
6           x = y-x;
7       else if (y > 10)
8           complexA();
9       else
10          x = x-y;
11          assert(0);
12  wait for t to complete;
}
```

```
bar(int b) {
13  y = b;
14  if (y>0)
15      x = x-y;
16      y = y-1;
17  else
18      complexB();
}
```

Fig. 4. A program with recursion and dynamically created threads

Assume the first executed trace is $\pi_1 = \langle(1, 0.1), (1, 1.1), (1, 2.1), (1, 3), (1, 0.2), (1, 1.2), (1, 2.2), (1, 5), (1, 6), (2, 13), (2, 14), (2, 15), (2, 16), (3, 13), (3, 14), (3, 15), (3, 16), (1, 12.1), (1, 12.2)\rangle$, in which Thread 1 creates Thread 2 and 3 that execute $bar(1)$. Note that in π_1 we drop the occurrence index if a statement of a thread occurs only once. An under-approximated symbolic analysis on π_1 does not yield an assertion violation, but the over-approximated symbolic analysis produces a counter-example $CEX_1 = \langle(1, 0), (1, 1), (2, 13), (2, 14), (2, 15), (1, 2), (1, 5), (1, 7), (1, 10), (1, 11)\rangle$, which leads to an assertion failure on Line 11. An execution following CEX_1 shows that the counterexample is spurious as it can only follow

up to $(1,5)$, because the else branch on Line 5 cannot be taken. The complete executed trace is $\pi_2 = \langle(1,0),(1,1),(2,13),(2,14),(2,15),(1,2),(1,5),(1,6),(2,16),(1,12)\rangle$. There is no assertion failure in π_2, but the counterexample obtained from the over-approximated analysis is $CEX_2 = \langle(1,0),(1,1),(2,13),(2,14),(2,15),(2,16),(1,2),(1,5),(1,7),(1,10),(1,11)\rangle$. A further execution is able to follow the complete trace of CEX_2 and therefore reveals a real assertion failure on line 11.

5 Optimizations

We apply *peephole partial order reduction* (PPOR) [29] to exploit the equivalence of interleavings due to independent transitions. Unlike classical partial order reduction [17,15], PPOR is able to reduce the search space symbolically in an SMT solver.

Given an executed trace $\pi = \langle(t_1, l_1.o_1, Q_1), \ldots (t_n, l_n.o_n, Q_n)\rangle$, we add a special scheduling constraint for every pair of tuples $(t_p, l_p.o_p, Q_p)$ and $(t_q, l_q.o_q, Q_q)$ such that $t_p \neq t_q$ and Q_p and Q_q are not dependent. Two statements are dependent if they access the same shared variable and at least one access is a write. For example, consider two statements $Q_p : a[k1] = e_1$ and $Q_q : a[k2] = e_2$ that are independent if the array index expressions do not have the same value. We add the following constraint to $\varphi(\pi)$:

$$L_p[i] = l_p.o_p \wedge L_q[i] = l_q.o_q \wedge k1|_{V \to V[i]} \neq k2|_{V \to V[i]} \to \neg(T[i] = q \wedge T[i+1] = p), \tag{12}$$

which prohibits Q_p being executed immediately after Q_q. Similar constraints can be added to over-approximated satisfiability formula $\psi(\pi)$.

Another optimization is a new thread-local static single assignment (TL-SSA) form to efficiently encode the thread-local statements. TL-SSA can significantly reduce the number of variables and the number of constraints needed in $\varphi(\pi)$ and $\psi(\pi)$, which are crucial since they often directly affect the performance of an SMT solver. Our observation is that the encoding in Section 3 may produce many redundant variables and constraints, due to the fact that it has to assign a fresh copy to every variable at every step. However, statements involving only local variables do not need a fresh copy of the local variables and constraints at every step. Furthermore, in a typical program execution, each statement writes to one variable at a time; a vast number of constraints, in the form of $v[i+1] = v[i]$, are used to keep the current values of the uninvolved variables.

In a purely sequential program, one can use Static Single Assignment (SSA) form [9] to simplify the encoding of a SAT formula [7]. However, SSA is not meant to be used in multithreaded programs (it remains an open problem as to what a SSA-style IR should be for concurrent programs), since a *use-define* chain for any shared variable cannot be established at compile time. Our observation here is that, while shared global variables cannot take advantages of the SSA form, local variables can still utilize the reduction power of SSA. The proposed TL-SSA form exploits the fact that, in any particular execution trace, the *use-define* chain of every *local* variable can be determined. Consider an executed trace

snippet $\langle\ldots$ (y=a+1), ..., (a=y), ..., (y=y+a)$\rangle$, where y is a shared variable and a is a local variables. In addition, no other statements in the trace access a. The trace with corresponding sequence of TL-SSA statements are $\langle\ldots$ (y=a_0+1),..., (a_1=y), ..., (y=y+a_1)$\rangle$. Instead of creating fresh copies for local variables at every step,the TL-SSA form creates only two copies of a. In addition, there is no need for the constraints $a[i+1] = a[i]$ to keep the value of a at each step where a is not assigned.

6 Related Work

Since we are not the first in modeling high-level source code semantics using a constraint language, it is helpful to briefly mention some of the successful approaches that have been reported. Noting the large gap between high-level programming languages and those of the formal logics, existing symbolic model checking tools, including [2,7,21], often restrict their representations to the pure Boolean domain; that is, they extract a Boolean-level model from the given program and then apply Binary Decision Diagrams (BDDs) [4] or SAT solvers (e.g., [11]) to perform verification. Although modeling all variables as bit-vectors is accurate, such high-precision approaches are often not needed and may generate models that are too large. In addition, bit vectors cannot model floating point arithmetic. In [32], sequential C programs are modeled at the word, as opposed to the bit, level using polyhedral analysis. This approach was shown to be very competitive for handling sequential C programs of non-trivial sizes. Unlike [32] that uses polyhedra library Omega [26] to perform reachability computation, we leverage the recently-demonstrated performance advances of SMT solvers to perform satisfiability checking.

Approaches based on similar ideas that augment testing with formal analysis include [20,18,6,24,28]. While Synergy [20] considers only sequential programs, we concentrate on multithreaded programs. Concolic testing [18,6,24] runs symbolic executions simultaneously with concrete executions, but the purpose is to generate new test inputs for better path coverage. In our approach, the purpose of symbolic analysis is to consider all related feasible thread interleavings implicitly, and in the event of inconclusive results, to guide the next concrete execution to follow a different thread schedule (that obeys program control flow semantics) under the same test vector. Predictive analysis [28] encodes a single execution symbolically without further refinement. The approach that augments testing with formal analysis has also been applied in other domains such as MCAPI [13,12] and service computing [14].

Although integrated under- and over-approximations have been used in a decision procedure [3] for bit-vector arithmetic, most previous works on hardware and software model checking follow the paradigm of CEGAR [22,8], which is based solely on over-approximations and uses spurious counterexamples to refine the over-approximations. In [19], Grumberg *et al.* presented a software model checking procedure based on a series of under-approximations.

Table 2. Bounded model checking (BMC) v.s trace-driven verification (TDV) for the multithreaded program in Fig. 4

	BMC		TDV		
#threads	mem(Mb)	time(s)	mem(Mb)	time(s)	speedup
5	21.15	1.16	20.14	1.07	1.08
10	54.29	3.18	51.92	3.15	1.01
15	129.11	66.01	100.72	7.64	8.64
20	219.34	169.84	166.81	18.69	9.09
25	317.40	215.87	250.13	44.33	4.87
30	420.54	222.85	348.18	45.83	4.86
35	538.75	140.21	461.85	42.11	3.33
40	692.62	150.66	597.55	73.69	2.04
45	-	-	745.86	77.79	-
50	-	-	906.9	143.27	-
55	-	-	1106.1	122.93	-
60	-	-	1330.5	182.53	-
65	-	-	1510.4	222.87	-
70	-	-	1737.5	289.86	-
75	-	-	2003.6	438.82	-
80	-	-	2270.1	407.07	-

There are several research projects that target concurrent program verification directly. Inspect [30] and CHESS [25] can check multithreaded C/C++ programs by explicitly executing different interleavings using dynamic partial order reduction [16]. However, explicitly exploring the thread interleavings does not scale well in the presence of a large number of (equivalence classes of) interleavings. The recent development in CHESS [25] also allows the tool to perform context bounded model checking. However, it is not intuitive to ask from user for a preset value on the number of context switches. CheckFence [5] checks all concurrent executions of a given C program on a relaxed memory model and verifies that they are observationally equivalent to a sequential execution, which targets a different application than ours.

7 Experiments

We have implemented a prototype of TDV using the Yices SMT solver [10], which is capable of deciding formulas with a combination of theories including propositional logic, linear arithmetic, and arrays. We performed two case studies. The first case study is on the example shown in Fig. 4 with multiple threads and recursions, and the second case study is on a file system implementation, which was previously used in [16]. Our experiments were conducted on a workstation with Pentium D 2.8 GHz CPU and 4GB memory running Red Hat Linux 7.2.

Table 2 shows the results of the first case study. By changing the value of the test variable a, we can increase the number of threads and the level of recursion. Column 1 lists the number of threads. Columns 2 and 3 show the peak memory

Table 3. Bounded model checking (BMC), trace-driven verification (TDV), and trace-driven verification with optimizations (TDVO)

filesystem example			BMC		TDV			TDVO		
#threads	depth	prop	mem	time	mem	time	speedup	mem	time	speedup
2	10	sat	13.51	34.8	8.86	15.7	2.22	7.58	1.7	20.47
2	16	sat	42.72	665.0	18.14	126.0	5.28	12.82	9.4	70.74
2	22	sat	40.56	2324.6	23.44	212.5	10.94	25.37	15.9	91.63
3	21	sat	194.21	49642.3	42.77	1823.1	27.23	30.95	381.8	130.02
2	10	unsat	7.50	7.2	5.36	1.03	6.99	5.27	0.26	27.69
2	16	unsat	15.85	824.9	13.37	82.7	9.97	7.76	1.24	665.24
3	15	unsat	73.07	9488.7	11.83	122.7	77.33	9.16	8.1	1171.44

and total time usage for Bounded Model Checking (BMC) without dynamic execution and abstraction. Columns 4 and 5 show the peak memory and total time usage for TDV. Note that optimizations has been applied to both methods. The last Column shows the speedup of the new method. A one-hour timeout limit is used in all the experiments. BMC ran out of time for test cases with more than 50 threads, while our method took only 407 seconds to complete 80 threads.

We also performed the experiments on the file system example, which is derived from a synchronization idiom found in the Frangipani file system. Table 3 shows the results we obtained by comparing BMC and TDV, both without and with optimizations. The results show that TDV gains a speedup from 1.46 to 77.33 over BMC, and the TDV with optimizations gains a speedup from 5.87 to 1171.44 over BMC, with an average speedup of 299.

8 Conclusion and Future Work

We have presented a new method to combine the efficiency of dynamic executions with the reasoning power of an SMT solver for the verification of safety properties of multithreaded programs. The main contributions are (1) a new symbolic encoding of executions of a multithreaded program, (2) using both under- and over-approximations in the same trace-driven abstraction framework, where refinement involving the mutual guidance between concrete program execution and symbolic analysis. For future work, we plan to investigate performance enhancement techniques, such as minimal unsatisfiable core analysis [23] and dynamic path reduction [31], to allow TDV to scale to larger programs.

References

1. Andraus, Z., Sakallah, K.: Automatic abstraction and verification of verilog models. In: Design Automation Conference (DAC), San Diego, California, pp. 218–223 (2004)

2. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.: Automatic predicate abstraction of c programs. In: Programming Language Design and Implementation, pp. 203–213 (2001)
3. Bryant, R., Kroening, D., Ouaknine, J., Seshia, S.A., Strichman, O., Brady, B.: Deciding bit-vector arithmetic with abstraction. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 358–372. Springer, Heidelberg (2007)
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers 35(8) (1986)
5. Burckhardt, S., Alur, R., Martin, M.M.K.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: Programming language Design and Implementation, pp. 12–21. ACM Press, New York (2007)
6. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: automatically generating inputs of death. In: ACM Conference on Computer and Communications Security. ACM, New York (2006)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
9. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
10. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
11. Een, N., Sorensson, N.: An extensible sat-solver. In: Satisfiability Workshop (2003)
12. Elwakil, M., Yang, Z.: Debugging Support Tool for MCAPI Applications. In: Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (2010)
13. Elwakil, M., Yang, Z., Wang, L.: CRI: Symbolic Debugger for MCAPI Applications. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 353–358. Springer, Heidelberg (2010)
14. Elwakil, M., Yang, Z., Wang, L., Chen, Q.: Message race detection for web services by an smt-based analysis. In: Sadjadi, S.M. (ed.) ATC 2010. LNCS, vol. 6407, pp. 182–194. Springer, Heidelberg (2010)
15. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Principles of Programming languages, pp. 110–121 (2005)
16. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. SIGPLAN Not. 40(1), 110–121 (2005)
17. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
18. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: PLDI, pp. 213–223 (2005)
19. Grumberg, O., Lerda, F., Strichman, O., Theobald, M.: Proof-guided underapproximation-widening for multi-process systems. SIGPLAN Notices 40(1), 122–131 (2005)
20. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: a new algorithm for property checking. In: Foundations of Software Engineering, pp. 117–127 (2006)

21. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M., Kahlon, V., Wang, C., Yang, Z.: Model checking C program using F-Soft. In: IEEE International Conference on Computer Design, San Jose, CA (October 2005)
22. Kurshan, R.P.: Computer-aided verification of coordinating processes. Princeton University Press, Princeton (1994)
23. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. 40(1), 1–33 (2008)
24. Majumdar, R., Sen, K.: Hybrid concolic testing. In: International Conference on Software Engineering. IEEE, Los Alamitos (2007)
25. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. SIGPLAN Not. 42(6), 446–455 (2007)
26. Pugh, W.: The omega test: a fast and practical integer programming algorithm for dependence analysis. In: ACM/IEEE Conference on Supercomputing, pp. 4–13. ACM, New York (1991)
27. Wang, C., Kim, H., Gupta, A.: Hybrid CEGAR: combining variable hiding and predicate abstraction. In: International Conference on Computer Aided Design, pp. 310–317 (2007)
28. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)
29. Wang, C., Yang, Z., Kahlon, V., Gupta, A.: Peephole partial order reduction. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 382–396. Springer, Heidelberg (2008)
30. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Runtime model checking of multithreaded C/C++ programs. Technical Report UUCS-07-008, School of Computing, University of Utah (2007)
31. Yang, Z., Al-Rawi, B., Sakallah, K., Huang, X., Smolka, S., Grosu, R.: Dynamic path reduction for software model checking. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423. Springer, Heidelberg (2009)
32. Yang, Z., Wang, C., Gupta, A., Ivanvčić, F.: Model checking sequential software programs via mixed symbolic analysis. ACM Transactions on Design Automation of Electronic Systems 14(1), 1–26 (2009)

Closed Form Approximations for Steady State Probabilities of a Controlled Fork-Join Network*

Jonathan Billington and Guy Edward Gallasch

Computer Systems Engineering Centre
School of Electrical and Information Engineering
University of South Australia
Mawson Lakes Campus, SA, 5095, Australia
{jonathan.billington,guy.gallasch}@unisa.edu.au

Abstract. Our work is motivated by just-in-time manufacturing systems, where goods are produced on demand. We consider a class of products made from two components each manufactured by its own production line. The components are then assembled, requiring synchronisation of the two lines. The production lines are coordinated to ensure that one line does not get ahead of the other by more than a certain number of components, N, a parameter of the system. We assume that the statistics of the processes follow exponential distributions, with requests to manufacture the product arriving at a rate λ_0 and the two production lines having rates λ_1 and λ_2. Generalised Stochastic Petri Nets (GSPN) are used to model this system where N is the initial marking of a control place. TimeNET is used to calculate the stationary token distribution of the GSPN as N increases, revealing convergence of the steady state probabilities. We characterise the range of rates for which useful convergence occurs using a large number of TimeNET runs and show how these results can be used to approximate the steady state probabilities for arbitrarily large N, to a desired level of accuracy. Further, for $\lambda_0 > \min(\lambda_1, \lambda_2)$ we discover geometric progressions in the steady state probabilities once they have converged. We use these progressions to derive closed form approximations, the accuracies of which increase as N increases.

Keywords: Fork-Join, Generalised Stochastic Petri Nets, Parametric Performance Analysis, Convergence, Closed Form Approximation, Geometric Progressions.

1 Introduction

We consider the computer control of a class of just in time manufacturing systems where goods are made from two different components (C1 and C2) based on customer demand. Everyday examples include hardware goods (e.g. hammers, screwdrivers, rakes, potato peelers). Each component is manufactured by its own production line, possibly by different manufacturers. The two production lines

* Supported by Australian Research Council Discovery Project DP0880928.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 420–435, 2010.
© Springer-Verlag Berlin Heidelberg 2010

need to be synchronised for assembly of the goods from the two components. The manufacturing system is driven by customer demand, so that no more goods are produced than those requested. Importantly, production is controlled so that each line can only be ahead of the other by a certain amount, to meet storage capacity constraints and to mitigate risk. Further, although the two types of components are different (e.g. a hammer's handle and head), components of the same type are identical, and hence the final product is made by assembling any available C1 component with any available C2 component.

Our goal is to analyse the performance of such systems to derive measures such as the average number of components one production line is ahead of another. However, the modelling of parallel systems where coordination and synchronisation occur requires the use of *fork-join* structures [1]. This presents significant challenges because product form solutions do not exist in general for systems with synchronisation. Due to their importance, fork-join systems have been studied by many authors (e.g. [2,3,4,5,1,6,7,8,9,10,11,12]). Basically these papers endeavour to find the mean response time of a job submitted to such systems, and often assume first come first served (FCFS) queueing disciplines. Our system is quite different. There is no identification of jobs (components) in our system, as any two of the different components can be assembled to form the same product. There is also no FCFS queueing discipline required for input or output. Moreover, in these papers, there is no concept of controlling the fork-join system to prevent one branch of the fork-join getting too far ahead of another. Our work has some similarities with the important class of assemble to order (ATO) systems [13], where several different products can be assembled from a number of components. However, our system is concerned with production of the components on demand, rather than their assembly on demand, and is more akin to (very simple) build to order (BTO) systems [14]. As far as we are aware the work on ATO systems (see [13] for an overview) does not handle the coordination of production lines to prevent too many of one component being produced.

We take advantage of the features of Generalised Stochastic Petri Nets (GSPNs) [15,16] to model this class of system. The two parallel production lines and their synchronisation are represented by a fork-join subnet. The fork-join subnet comprises two parallel branches where each branch includes a transition with a firing time governed by the exponential distribution. There is a further exponential transition representing requests, that feeds the fork-join. The environment of the fork-join ensures that one line can never be ahead of the other by more than N components. We can then use steady state Markov analysis [15,16] to derive performance measures for this system. Since we wish to characterise the system's behaviour as N is varied, we treat N as a positive integer parameter of the system. Earlier work [17,18] was concerned with aggregating the fork-join subnet. Unfortunately, these results are not applicable for our work, since aggregation does not allow any performance measures to be calculated within the fork-join, which is our goal.

Since finding exact closed form solutions for arbitrary N is very difficult [19] due to the fork-join, this paper is concerned with approximate solutions. Very

recently [20] we calculated the steady state probabilities of the GSPN using TimeNET [21] over a wide range of transition rates, and for N up to 50, which revealed convergence as N increases. We denoted the value of N by which convergence occurred to dp decimal places by N_{cdp} and showed how N_{cdp} varies as a function of ratios of the rates and dp. This allowed us to approximate the probabilities for arbitrary $N > N_{cdp}$ to a certain accuracy within the useful convergence region. In [20], we also derived a heuristic to estimate N_{cdp}.

This paper builds on the results in [20]. Firstly, results in this paper are more comprehensive, incorporating values of N up to 100, and additional values of the rates. We thus provide a better characterisation of convergence which allows its useful range to be extended. Secondly, our previous results suffered from *rounding anomalies* [20], which led to 'bumps' in the graphs and failures in the heuristic (see Figs. 3 and 4 and Table 3 of [20]). This is overcome in this paper by redefining N_{cdp} to be the value of N by which the probabilities are within 0.5×10^{-dp} of the converged value, which is now estimated by the probability when $N = 100$ (rather than 50). Thirdly, this paper provides a much better demonstration that N_{cdp} increases approximately linearly with dp, and uses a different example with two dimensional graphs to better illustrate the results. Finally, in contrast with [20], we do not consider heuristics for determining the value of N_{cdp}. Instead we concentrate on providing closed form approximate solutions. When the rate of customer requests exceeds the minimum production line rate, geometric progressions exist in the converged steady state probabilities. We use these progressions to derive closed form approximate solutions for the probabilities in terms of the rates and the capacity N. Importantly, the accuracy of the approximation increases as N increases, allowing solutions to be obtained for plants with very large capacities.

The structure of the paper is as follows. Section 2 presents the parametric GSPN model, while its associated family of continuous time Markov chains is discussed in Section 3. Section 4 characterises the convergence of the steady state probabilities. Approximate closed form solutions are derived in Section 5, which also discusses their accuracy. Conclusions are drawn in Section 6. Some familiarity with GSPNs and their analysis [15,16] is assumed.

2 Parametric GSPN Model

We consider the parametric GSPN, $GSPN_N$, shown in Fig. 1, which is essentially the same as in [17]. $GSPN_N$ includes a fork-join subnet with 2 branches between immediate transitions, t_1 (the fork) and t_2 (the join). The remaining transitions are exponentially distributed timed transitions with their own rates, i.e. λ_i is associated with T_i, $i \in \{0, 1, 2\}$. The left branch of the subnet (P_1, T_1, P_3) represents the production of component C1 (stored in P_3) and the right (P_2, T_2, P_4) the production of C2 (stored in P_4). Place P_0 controls the capacity of the production lines. Initially all places are empty except for P_0 which contains N tokens, restricting the capacity of the lines to N components. T_0 represents customer requests. Each request results in 1 unit of the raw materials needed to create

components C1 and C2 being deposited in places P_1 and P_2 respectively (via the fork). P_1 and P_2 represent stores for the raw material, which is not identified nor related to a particular request. For efficiency, as soon as both components are available they are removed immediately (transition t_2) for assembly, freeing up capacity in the lines. This is modelled by a token being added to P_0 indicating that there is now additional capacity for each component. $GSPN_N$ ensures that the same number of components (C1 and C2) is produced as goods requested, eliminating waste, and that they are produced on demand. It also ensures that no more than N C1 components can be produced in advance of a C2 component and vice versa, providing the desired level of coordination between the production lines.

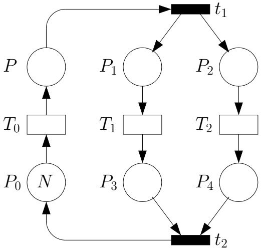


Fig. 1. $GSPN_N$ with a Fork-Join Subnet

3 Family of Continuous Time Markov Chains

We would like to obtain steady state solutions for the probability of being in a particular marking of the GSPN. Unfortunately, obtaining analytical solutions for the steady state probabilities is a difficult problem for arbitrary N, due to the lack of product form solutions. We thus turn to numerical solutions using TimeNET [21], which can calculate the steady state probabilities directly from the GSPN. However, to discuss the results, we firstly provide some insight into the structure of the family of continuous time Markov chains (CTMCs) that are associated with $GSPN_N$, and define some notation for them.

The family of CTMCs, $CTMC_N$, can be derived from $GSPN_N$ by generating its reduced reachability graph [15,16]. We depict $CTMC_N$ for $N = 1, 2, 3$ in Fig. 2. We can observe that the number of states in $CTMC_N$ is $(N + 1)^2$. We number the states from 1 to $(N + 1)^2$ in vertical columns, from left to right and top to bottom. The initial marking is mapped to state 1, the top right vertex is state $N^2 + 1$, and the bottom right vertex is state $(N + 1)^2$. Note the symmetry about the horizontal centre row. Finally we denote the steady state probability of state i of $CTMC_N$ by $\pi_{(N,i)}, 1 \leq i \leq (N + 1)^2$.

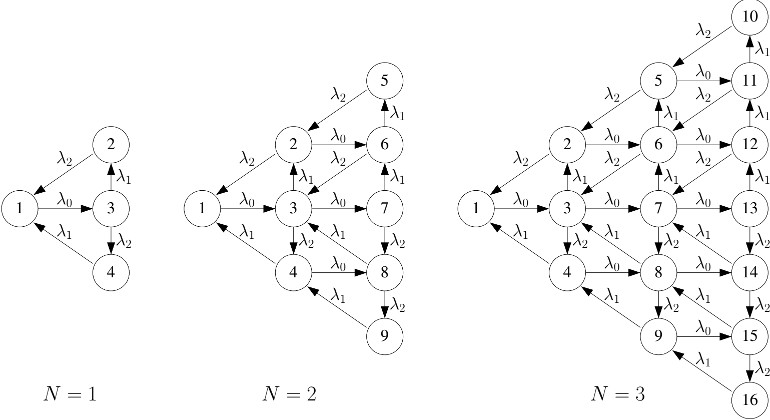


Fig. 2. $CTMC_N$ for $N = 1$, 2 and 3

4 Characterising Convergence

Using TimeNET, we have performed steady state analysis of $GSPN_N$ for N from 1 to 100 for a wide range of firing rates, which we call *configurations*. Inspection of the results has identified powerful convergence trends that allow the approximation of the steady state probabilities for arbitrary N with excellent accuracy, based on the probabilities for moderately large N.

Because the results depend on ratios of the firing rates, and taking into account the symmetry between λ_1 and λ_2, we fixed λ_1 to 1 while varying λ_2 and λ_0. For N from 1 to 100 we analysed 448 different configurations in which λ_2 varied over 1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.7, 2, 3, 4, 5, 8, 10, 20, 50 and 100, and λ_0 over 0.01, 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1, 1.2, 1.3, 1.4, 1.5, 1.7, 2, 3, 4, 5, 8, 10, 20, 50 and 100, a total of 44800 TimeNET runs. We chose these values due to convergence being less rapid as the rates approach each other, (i.e. as the ratios approach 1). Because of the dependence on ratios of firing rates, these results can be scaled to an unbounded number of configurations that maintain the ratios, $\frac{\lambda_0}{\lambda_1}$ and $\frac{\lambda_2}{\lambda_1}$.

4.1 Convergence When $\lambda_0 < \min(\lambda_1, \lambda_2)$

Consider the example of $\lambda_0 = 0.2$, $\lambda_1 = 1$ and $\lambda_2 = 2$. Tables 1 and 2 present the steady state probabilities (to 4 decimal places) of all four states of the CTMC when $N = 1$, all 9 states when $N = 2$, all 16 states when $N = 3$, and the first 16 states (i.e. the first 4 columns of the CTMC) for N from 4 to 10 and for $N = 100$. Steady state probabilities increase to the left of the CTMC due to λ_0 being smaller than both λ_1 and λ_2, and are biased toward the states in the lower half of the CTMC (e.g. $\pi_{(N,4)} > \pi_{(N,2)}$) due to $\lambda_2 > \lambda_1$. The reverse bias would exist if $\lambda_1 > \lambda_2$ due to the symmetry of the CTMC.

Table 1. Convergence of $\pi_{(N,1)}$ to $\pi_{(N,9)}$ as N increases for $\lambda_0=0.2$, $\lambda_1=1$ and $\lambda_2=2$

	$\pi_{(N,1)}$	$\pi_{(N,2)}$	$\pi_{(N,3)}$	$\pi_{(N,4)}$	$\pi_{(N,5)}$	$\pi_{(N,6)}$	$\pi_{(N,7)}$	$\pi_{(N,8)}$	$\pi_{(N,9)}$
N=1	0.8108	0.0270	0.0541	0.1081					
N=2	0.7801	0.0256	0.0535	0.1048	0.0014	0.0029	0.0036	0.0094	0.0187
N=3	0.7745	0.0254	0.0531	0.1041	0.0014	0.0029	0.0037	0.0094	0.0186
N=4	0.7734	0.0254	0.0531	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=5	0.7732	0.0254	0.0531	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=6	0.7732	0.0254	0.0530	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=7	0.7732	0.0254	0.0530	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=8	0.7732	0.0254	0.0530	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=9	0.7732	0.0254	0.0530	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=10	0.7732	0.0254	0.0530	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186
N=100	0.7732	0.0254	0.0530	0.1039	0.0014	0.0029	0.0037	0.0094	0.0186

Table 2. Convergence of $\pi_{(N,10)}$ to $\pi_{(N,16)}$ as N increases for $\lambda_0 = 0.2$, $\lambda_1 = 1$ and $\lambda_2=2$

	$\pi_{(N,10)}$	$\pi_{(N,11)}$	$\pi_{(N,12)}$	$\pi_{(N,13)}$	$\pi_{(N,14)}$	$\pi_{(N,15)}$	$\pi_{(N,16)}$
N=1							
N=2							
N=3	0.0001	0.0002	0.0003	0.0002	0.0008	0.0018	0.0035
N=4	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=5	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=6	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=7	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=8	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=9	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=10	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035
N=100	0.0001	0.0002	0.0003	0.0003	0.0008	0.0018	0.0035

When considering convergence of these probabilities to a given number of decimal places dp, we require that **all** probabilities must be within $\pm 0.5 \times 10^{-dp}$ of their final converged value. Taking the probabilities obtained for $N = 100$ as a close approximation of their converged values, we can see from Table 1 that, for example, $\pi_{(N,1)}$ converges to within 0.5×10^{-3} of its final value by $N = 4$. For this configuration, all probabilities have converged to within 0.5×10^{-3} of their final values by $N = 4$, including $\pi_{(N,17)}$ to $\pi_{(N,(N+1)^2)}$ not shown in these two tables where for $5 \le N \le 100$ all additional states (that don't exist when $N = 4$) have probabilities less than 0.5×10^{-3} (effectively zero for this degree of accuracy). Hence, for this configuration, these convergence results mean that the steady state probabilities can be approximated to three decimal places for arbitrary $N > 4$ for states 1 to 25 by $\pi_{(4,1)}$ to $\pi_{(4,25)}$ and by 0 for states 26 to $(N + 1)^2$.

We can now generalise this result for any number of decimal places. Let N_{cdp} be the minimum value of N by which convergence has occurred to within 0.5×10^{-dp} with all additional states for $N > N_{cdp}$ having probabilities less than

0.5×10^{-dp}. Then for arbitrary $N > N_{cdp}$ the probabilities are given to within 0.5×10^{-dp} of their final converged values by:

1. $\pi_{(N,i)} = \pi_{(N_{cdp},i)}$ for $1 \leq i \leq (N_{cdp}+1)^2$; and
2. $\pi_{(N,i)} = 0$ for $(N_{cdp}+1)^2 < i \leq (N+1)^2$.

For example, for the considered configuration, $N_{c3} = 4$ (as described above), $N_{c4} = 6$ and $N_{c5} = 7$. We discuss the practicality of this convergence result in section 4.3.

4.2 Convergence When $\lambda_0 > \min(\lambda_1, \lambda_2)$

For this scenario, $\lambda_1 = 1$ and $\lambda_2 = 2$ as before, but now $\lambda_0 = 5$, so that λ_0 is a factor of 5 larger than $\min(\lambda_1, \lambda_2)$ instead of a factor of 5 smaller. In this situation the steady state probabilities increase toward the right edge of the CTMC due to $\lambda_0 > \min(\lambda_1, \lambda_2)$, and toward the lower edge of the CTMC due to $\lambda_2 > \lambda_1$.

Table 3 presents the steady state probabilities (to 4 decimal places) of the states in the right-most vertical column of $CTMC_N$, from the lower edge (state $(N+1)^2$) towards the upper edge (state N^2+1) as N increases. Tables 4 and 5 do the same for the states in the second and third vertical columns from the right respectively. These probabilities (and those not shown in Tables 3 to 5) converge to within 0.5×10^{-3} of their final values by $N_{c3} = 7$. Hence it is possible to approximate the steady state probabilities for arbitrary $N > 7$, to 3 decimal places, using the values for $N_{c3} = 7$, however it is not as straightforward as it was when $\lambda_0 < \min(\lambda_1, \lambda_2)$. Because probabilities increase toward the lower right vertex of the CTMC and not the left vertex of the CTMC, the converged probabilities are associated with states that are relative to the lower right vertex, $(N+1)^2$, (rather than state 1, the left vertex) as N increases. Hence, the probabilities of the lower right states of $CTMC_N$, $N > N_{cdp}$, are approximated by the probabilities of the states in $CTMC_{N_{cdp}}$. For this example, $\pi_{(N,(N+1)^2-14)}$ to $\pi_{(N,(N+1)^2)}$ for $N > 7$ can be approximated by $\pi_{(7,50)}$ to $\pi_{(7,64)}$ (from an extended Table 3), $\pi_{(N,N^2-12)}$ to $\pi_{(N,N^2)}$ can be approximated by $\pi_{(7,37)}$ to $\pi_{(7,49)}$ (from an extended Table 4), $\pi_{(N,(N-1)^2-10)}$ to $\pi_{(N,(N-1)^2)}$ can be approximated by $\pi_{(7,26)}$ to $\pi_{(7,36)}$(from Table 5), $\pi_{(N,(N-2)^2-8)}$ to $\pi_{(N,(N-2)^2)}$ can be approximated by $\pi_{(7,17)}$ to $\pi_{(7,25)}$, $\pi_{(N,(N-3)^2-6)}$ to $\pi_{(N,(N-3)^2)}$ can be approximated by $\pi_{(7,10)}$ to $\pi_{(7,16)}$, $\pi_{(N,(N-4)^2-4)}$ to $\pi_{(N,(N-4)^2)}$ can be approximated by $\pi_{(7,5)}$ to $\pi_{(7,9)}$, $\pi_{(N,(N-5)^2-2)}$ to $\pi_{(N,(N-5)^2)}$ can be approximated by $\pi_{(7,2)}$ to $\pi_{(7,4)}$ and $\pi_{(N,(N-6)^2)}$ can be approximated by $\pi_{(7,1)}$. The remaining state probabilities are approximated by zero. This can be generalised to the following approximation for arbitrary $N > N_{cdp}$:

1. $\pi_{(N,(N-x+1)^2-y)} = \pi_{(N_{cdp},(N_{cdp}-x+1)^2-y)}$, for $0 \leq x \leq N_{cdp}$ and $0 \leq y \leq 2(N_{cdp}-x)$; and
2. $\pi_{(N,i)} = 0$ for all other states, i.e. for $1 \leq i \leq (N-N_{cdp})^2$ and $(N-x)^2+1 \leq i \leq (N-x)^2 + 2(N-N_{cdp})$ where $0 \leq x \leq N_{cdp}$.

where this essentially translates the probabilities of $CTMC_{N_{cdp}}$ to the bottom right corner of $CTMC_N$, with the remaining probabilities equal to zero.

Table 3. Convergence of $\pi_{(N,(N+1)^2)}$ to $\pi_{(N,N^2+1)}$ as N increases for $\lambda_0 = 5$, $\lambda_1 = 1$ and $\lambda_2 = 2$ (A denotes $(N+1)^2$)

	$\pi_{(N,A)}$	$\pi_{(N,A-1)}$	$\pi_{(N,A-2)}$	$\pi_{(N,A-3)}$	$\pi_{(N,A-4)}$	$\pi_{(N,A-5)}$	$\pi_{(N,A-6)}$	$\pi_{(N,A-7)}$	$\pi_{(N,A-8)}$
N=1	0.4878	0.2439	0.1220						
N=2	0.4216	0.2108	0.0937	0.0586	0.0293				
N=3	0.4057	0.2029	0.0998	0.0434	0.0288	0.0153	0.0076		
N=4	0.4015	0.2007	0.1001	0.0491	0.0213	0.0144	0.0078	0.0040	0.0020
N=5	0.4004	0.2002	0.1001	0.0499	0.0244	0.0106	0.0072	0.0039	0.0020
N=6	0.4001	0.2000	0.1000	0.0500	0.0249	0.0122	0.0053	0.0036	0.0020
N=7	0.4000	0.2000	0.1000	0.0500	0.0250	0.0125	0.0061	0.0026	0.0018
N=8	0.4000	0.2000	0.1000	0.0500	0.0250	0.0125	0.0062	0.0031	0.0013
N=9	0.4000	0.2000	0.1000	0.0500	0.0250	0.0125	0.0062	0.0031	0.0015
N=10	0.4000	0.2000	0.1000	0.0500	0.0250	0.0125	0.0062	0.0031	0.0016
N=100	0.4000	0.2000	0.1000	0.0500	0.0250	0.0125	0.0063	0.0031	0.0016

Table 4. Convergence of $\pi_{(N,N^2)}$ to $\pi_{(N,(N-1)^2+1)}$ as N increases for $\lambda_0 = 5$, $\lambda_1 = 1$ and $\lambda_2 = 2$ (A denotes N^2)

	$\pi_{(N,A)}$	$\pi_{(N,A-1)}$	$\pi_{(N,A-2)}$	$\pi_{(N,A-3)}$	$\pi_{(N,A-4)}$	$\pi_{(N,A-5)}$	$\pi_{(N,A-6)}$	$\pi_{(N,A-7)}$	$\pi_{(N,A-8)}$
N=1	0.1463								
N=2	0.0890	0.0562	0.0164						
N=3	0.0818	0.0425	0.0261	0.0086	0.0034				
N=4	0.0804	0.0404	0.0209	0.0128	0.0044	0.0018	0.0008		
N=5	0.0801	0.0401	0.0202	0.0104	0.0064	0.0022	0.0009	0.0004	0.0002
N=6	0.0800	0.0400	0.0200	0.0101	0.0052	0.0032	0.0011	0.0005	0.0002
N=7	0.0800	0.0400	0.0200	0.0100	0.0050	0.0026	0.0016	0.0006	0.0002
N=8	0.0800	0.0400	0.0200	0.0100	0.0050	0.0025	0.0013	0.0008	0.0003
N=9	0.0800	0.0400	0.0200	0.0100	0.0050	0.0025	0.0013	0.0007	0.0004
N=10	0.0800	0.0400	0.0200	0.0100	0.0050	0.0025	0.0013	0.0006	0.0003
N=100	0.0800	0.0400	0.0200	0.0100	0.0050	0.0025	0.0013	0.0006	0.0003

Table 5. Convergence of $\pi_{(N,(N-1)^2)}$ to $\pi_{(N,(N-2)^2+1)}$ as N increases for $\lambda_0 = 5$, $\lambda_1 = 1$ and $\lambda_2 = 2$ (A denotes $(N-1)^2$)

	$\pi_{(N,A)}$	$\pi_{(N,A-1)}$	$\pi_{(N,A-2)}$	$\pi_{(N,A-3)}$	$\pi_{(N,A-4)}$	$\pi_{(N,A-5)}$	$\pi_{(N,A-6)}$	$\pi_{(N,A-7)}$	$\pi_{(N,A-8)}$
N=1									
N=2	0.0244								
N=3	0.0170	0.0102	0.0024						
N=4	0.0162	0.0084	0.0049	0.0013	0.0004				
N=5	0.0160	0.0081	0.0042	0.0024	0.0007	0.0002	0.0001		
N=6	0.0160	0.0080	0.0040	0.0021	0.0012	0.0003	0.0001	0.0000	0.0000
N=7	0.0160	0.0080	0.0040	0.0020	0.0010	0.0006	0.0002	0.0001	0.0000
N=8	0.0160	0.0080	0.0040	0.0020	0.0010	0.0005	0.0003	0.0001	0.0000
N=9	0.0160	0.0080	0.0040	0.0020	0.0010	0.0005	0.0003	0.0002	0.0000
N=10	0.0160	0.0080	0.0040	0.0020	0.0010	0.0005	0.0003	0.0001	0.0001
N=100	0.0160	0.0080	0.0040	0.0020	0.0010	0.0005	0.0003	0.0001	0.0001

4.3 Characterisation of Convergence

The above convergence experiments have been repeated for all configurations and for accuracies of $0.5 \times 10^{-dp}, 1 \leq dp \leq 10$. For $dp = 3$ the values of N_{c3} are depicted in Fig. 3 for $\frac{\lambda_2}{\lambda_1} = 1.1, 1.2, 1.3, 1.4, 1.5, 1.7, 2, 5, 10$ and 100. The x-axis of this graph uses a log scale, to depict $\frac{\lambda_0}{\lambda_1}$ from 0.01 to 100. $\lambda_0 = 1$ is hence positioned at the centre of the x-axis. A clear trend evident from this graph is for convergence to slow as λ_0 approaches $\min(\lambda_1, \lambda_2) = 1$ from both below and above.

When $\frac{\lambda_0}{\lambda_1}$ approaches 1 from below, the speed of convergence varies significantly with λ_0 but exhibits little or no variation when varying λ_2 (the curves sit almost on top of each other). However, for $\frac{\lambda_0}{\lambda_1} > 1$, the value of λ_2 has a significant impact. For $\frac{\lambda_2}{\lambda_1} = 100$, the speed of convergence as $\frac{\lambda_0}{\lambda_1}$ increases from 1 to 100 follows a trend that is the mirror of the trend as $\frac{\lambda_0}{\lambda_1}$ decreases from 1 to 0.01. However, for decreasing $\frac{\lambda_2}{\lambda_1}$, the speed of convergence diverges from this trend at increasingly smaller $\frac{\lambda_0}{\lambda_1}$ values. For example, for $\frac{\lambda_2}{\lambda_1} = 1.1$, the divergence occurs at around $\frac{\lambda_0}{\lambda_1} = 1.2$. The values of N_{c3} then increase from 31 ($\frac{\lambda_0}{\lambda_1} = 1.2$), to 38 ($\frac{\lambda_0}{\lambda_1} = 2$) before decreasing to 27 ($\frac{\lambda_0}{\lambda_1} = 20$) and appearing to remain constant as $\frac{\lambda_0}{\lambda_1}$ increases to 100. It is clear that the severity of the deviation increases as λ_2 approaches λ_1. The point at which the deviation occurs is less clear, although we observe that it appears to roughly coincide with the point at which λ_0 becomes larger than λ_2.

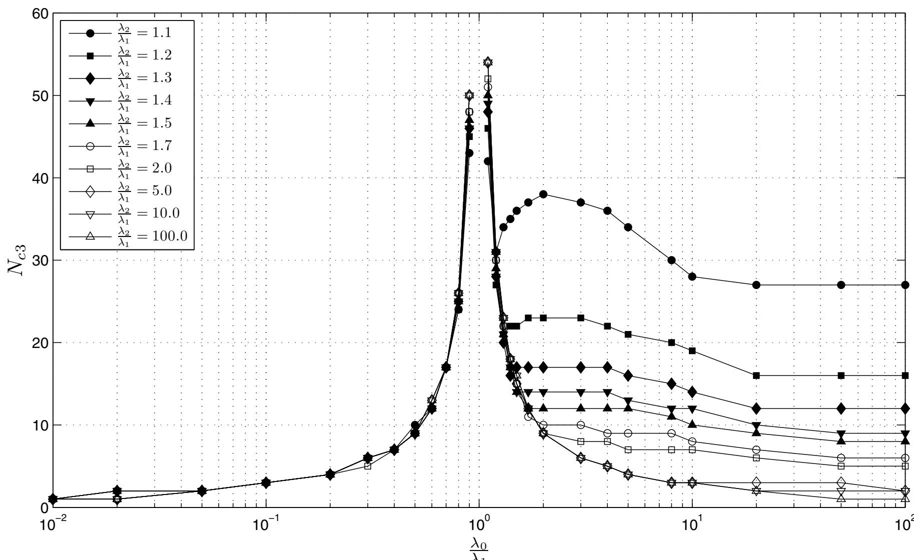


Fig. 3. N_{c3} values when varying λ_0 and λ_2

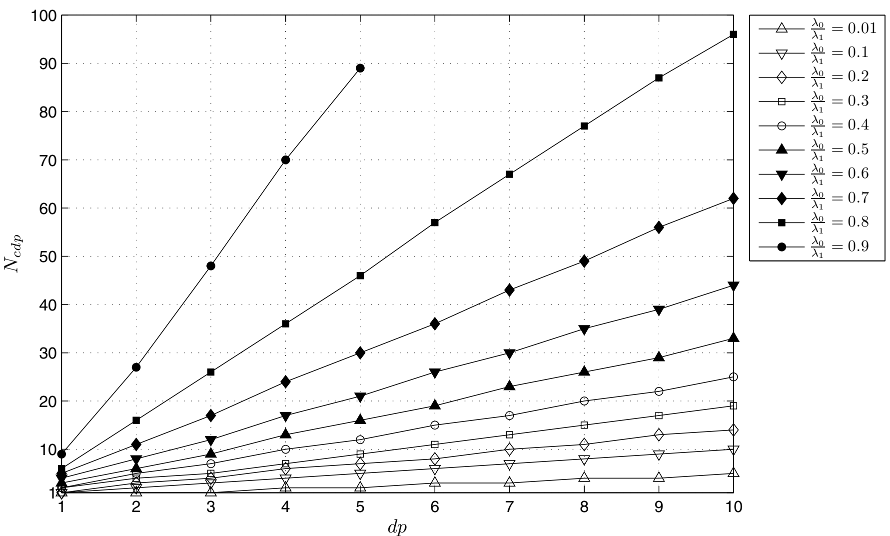


Fig. 4. N_{cdp} for varying dp from 1 to 10, and varying λ_0 from 0.01 to 0.9, with $\lambda_2 = 2$

The case of $\frac{\lambda_2}{\lambda_1} = 1$ is not depicted in Fig. 3. When $\frac{\lambda_2}{\lambda_1} = 1$ we observed that, for $\lambda_0 < \lambda_1$, the corresponding values of N_{c3} followed the same trend as all other $\frac{\lambda_2}{\lambda_1}$ ratios depicted in the left half of Fig. 3. However, when $\lambda_0 > \lambda_1$, convergence of steady state probabilities to within 0.5×10^{-3} of their final converged values did not occur for $N \leq 100$.

The graph in Fig. 4 depicts N_{cdp} for $1 \leq dp \leq 10$ and selected values of λ_0 from 0.01 to 0.9, with $\lambda_2 = 2$. We observe that N_{cdp} increases approximately linearly in dp. This trend is also evident for λ_0 from 1.1 to 100 (not depicted in Fig. 4). Again, convergence slows as λ_0 approaches λ_1, as observed for N_{c3} in Fig. 3. This linear trend, when extrapolated, allows prediction of N_{cdp} for any $dp > 10$.

From these observations, it is possible to characterise the ranges of ratios of rates for which useful convergence results can be obtained:

- $\frac{\lambda_0}{\lambda_1} \leq\approx 0.9$ and any value of $\frac{\lambda_2}{\lambda_1} \geq 1$, and
- $\frac{\lambda_0}{\lambda_1} \geq\approx 1.1$ and $\frac{\lambda_2}{\lambda_1} \geq\approx 1.1$;

where the convergence results become more accurate (i.e. can be determined for larger dp values) as λ_0 becomes increasingly smaller than λ_1, or $\lambda_0 > \lambda_1$ and λ_0 or λ_2 become increasingly larger than λ_1.

5 Closed Form Approximate Probabilities

In this section we derive closed form expressions for the steady state probabilities based on the observation that, for $\lambda_0 > \min(\lambda_1, \lambda_2)$, when the probabilities have converged they follow geometric progressions.

5.1 Approximate Probabilities When $\lambda_0 > \min(\lambda_1, \lambda_2)$

From Tables 3, 4 and 5 we can see that the probabilities follow two geometric progressions once they have converged. The first has a common ratio of $\frac{\lambda_1}{\lambda_2}$ from bottom to top of each vertical column of states in each CTMC. For example, in each of Tables 3, 4 and 5, we see that each entry from left to right in the row for $N = 100$ is multiplied by $\frac{\lambda_1}{\lambda_2} = 0.5$. The second geometric progression is along the bottom boundary of the CTMC, from $\pi_{(N,(N+1)^2)}$ to $\pi_{(N,1)}$, where the common ratio is $\frac{\lambda_1}{\lambda_0} = 0.2$ as can be seen in Tables 3, 4 and 5 by comparing the corresponding entries in each table when $N = 100$. This is the case for all our results when $\lambda_0 > \min(\lambda_1, \lambda_2)$. Hence, once the probabilities have converged we can express each probability in terms of $\pi_{(N,(N+1)^2)}$, using these geometric progressions. For $N \geq N_{cdp}, \lambda_0 > \lambda_1, \lambda_2 > \lambda_1$, we have:

$$\pi_{(N,(x+1)^2-y)} = \left(\frac{\lambda_1}{\lambda_0}\right)^{N-x} \left(\frac{\lambda_1}{\lambda_2}\right)^{y} \pi_{(N,(N+1)^2)}, 0 \leq x \leq N \text{ and } 0 \leq y \leq 2x \quad (1)$$

Because the sum of these probabilities equals 1, we can determine $\pi_{(N,(N+1)^2)}$. Let $r_{10} = \frac{\lambda_1}{\lambda_0}$ and $r_{12} = \frac{\lambda_1}{\lambda_2}$ in equation (1), then:

$$\sum_{x=0}^{N}\sum_{y=0}^{2x} {r_{10}}^{N-x} {r_{12}}^{y} \pi_{(N,(N+1)^2)} = 1$$

$$\Rightarrow \pi_{(N,(N+1)^2)} \sum_{x=0}^{N} {r_{10}}^{N-x} \sum_{y=0}^{2x} {r_{12}}^{y} = 1$$

$$\Rightarrow \pi_{(N,(N+1)^2)} \sum_{x=0}^{N} {r_{10}}^{N-x} \left(\frac{1-{r_{12}}^{2x+1}}{1-r_{12}}\right) = 1$$

$$\Rightarrow \pi_{(N,(N+1)^2)} \sum_{x=0}^{N} {r_{10}}^{N-x}(1-{r_{12}}^{2x+1}) = 1 - r_{12}$$

$$\Rightarrow \pi_{(N,(N+1)^2)} = \frac{1-r_{12}}{Sum} \quad (2)$$

where

$$\begin{aligned} Sum &= \sum_{x=0}^{N} {r_{10}}^{N-x}(1-{r_{12}}^{2x+1}) \\ &= \sum_{x=0}^{N} {r_{10}}^{x} - r_{12}{r_{10}}^{N} \sum_{x=0}^{N} \left(\frac{{r_{12}}^2}{r_{10}}\right)^{x} \\ &= \frac{1-{r_{10}}^{N+1}}{1-r_{10}} - r_{12}{r_{10}}^{N} SumB \end{aligned}$$

with

$$SumB = \sum_{x=0}^{N} \left(\frac{{r_{12}}^2}{r_{10}}\right)^x = \begin{cases} N+1, & r_{10} = {r_{12}}^2 \\ \frac{1-\left(\frac{{r_{12}}^2}{r_{10}}\right)^{N+1}}{1-\frac{{r_{12}}^2}{r_{10}}}, & otherwise \end{cases}$$

so that

$$Sum = \frac{1-{r_{10}}^{N+1}}{1-r_{10}} - \begin{cases} r_{12}(N+1){r_{10}}^N, & r_{10} = {r_{12}}^2 \\ r_{12}\left(\frac{{r_{10}}^{N+1}-{r_{12}}^{2(N+1)}}{r_{10}-{r_{12}}^2}\right), & otherwise \end{cases}$$

Because we are interested in the result when the probabilities have converged, we take the limit of Sum as $N \to \infty$. As $|r_{10}| < 1$ and $|r_{12}| < 1$, ${r_{10}}^{N+1}$, ${r_{12}}^{2(N+1)}$ and $(N+1){r_{10}}^N$ all tend to 0 as $N \to \infty$, and thus

$$\lim_{N\to\infty} Sum = \frac{1}{1-r_{10}} \tag{3}$$

Using equation (3) in equation (2) yields

$$\lim_{N\to\infty} \pi_{(N,(N+1)^2)} = (1-r_{10})(1-r_{12}) = \left(1-\frac{\lambda_1}{\lambda_0}\right)\left(1-\frac{\lambda_1}{\lambda_2}\right)$$

Thus for $\lambda_0 > \lambda_1$ and $\lambda_2 > \lambda_1$, we can approximate the steady state probabilities by using this result in equation (1).

$$\pi_{(N,(x+1)^2-y)} \approx \left(\frac{\lambda_1}{\lambda_0}\right)^{N-x} \left(\frac{\lambda_1}{\lambda_2}\right)^y \left(1-\frac{\lambda_1}{\lambda_0}\right)\left(1-\frac{\lambda_1}{\lambda_2}\right), \quad 0 \le x \le N, 0 \le y \le 2x \tag{4}$$

Similarly, due to symmetry, for $\lambda_0 > \lambda_2, \lambda_1 > \lambda_2$, the steady state probabilities are approximated by

$$\pi_{(N,(x^2+1)+y)} \approx \left(\frac{\lambda_2}{\lambda_0}\right)^{N-x} \left(\frac{\lambda_2}{\lambda_1}\right)^y \left(1-\frac{\lambda_2}{\lambda_0}\right)\left(1-\frac{\lambda_2}{\lambda_1}\right), \quad 0 \le x \le N, 0 \le y \le 2x \tag{5}$$

5.2 Accuracy of the Approximation

We expect that the approximation will not be particularly good for small N but will become progressively better as N increases (i.e. as the probabilities converge).

We firstly compare the approximation with Table 3. The first column of this table gives the probabilities for $\pi_{(N,(N+1)^2)}$ for N up to 10 and the value for $N = 100$. From equation (4), $\pi_{(N,(N+1)^2)} \approx (1-\frac{\lambda_1}{\lambda_0})(1-\frac{\lambda_1}{\lambda_2}) = (1-0.2)(1-0.5) = 0.4$. This is the same as the value for $N > 6$ from the table. This is expected,

as the probabilities converge to $dp = 3$, by $N = 7$. The approximate value for $\pi_{(N,(N+1)^2-1)}$ is half that of $\pi_{(N,(N+1)^2)}$, i.e. 0.2, which is again the value for $N > 6$, but this time it is also a very good approximation for $N = 6$. This trend is evident for the rest of the probabilities in Table 3. However, for the probabilities in the last two columns, the approximation is not so good for $N = 7$, but still within $.5 \times 10^{-3}$, which is expected when $dp = 3$. However, for $N > 9$, the approximation is again excellent and accurate to 4 decimal places. Similar trends occur in Tables 4 and 5. Note also from equation (4), that $\pi_{(N,N^2)} \approx \frac{\lambda_1}{\lambda_0} \times \pi_{(N,(N+1)^2)} = 0.2 \times 0.4 = 0.08$, which is the converged value in Table 4, as expected, and a similar result occurs when moving from Table 4 to Table 5.

To evaluate the accuracy of the approximation more broadly, we took the 225 configurations in which $\lambda_0 > \lambda_1$ and $\lambda_2 > \lambda_1$, calculated the approximate probabilities given by equation (4) for N from 1 to 100, and compared the approximate values to the results obtained from TimeNET. We denote by N_{adp} the smallest value of N by which all the approximate probabilities for a particular configuration are accurate to within 0.5×10^{-dp} of the TimeNET results for $1 \leq dp \leq 10$. 49 of the configurations are shown in Table 6. A dash in the table indicates that the required accuracy for the value of dp was not reached by $N = 100$. These results show that the approximation achieves the specified accuracy at a value of N that is the same or less than N_{cdp}. In some cases the difference is quite significant. For example, when $\lambda_0 = 1.5$, $\lambda_2 = 1.1$ and $dp = 4$, the approximation achieves the desired accuracy by $N = 44$, whereas convergence does not occur until $N = 61$. This means that for this configuration for an accuracy of $dp = 4$, that TimeNET results should be used for $1 \leq N < 44$, whereas the approximation can be used for $N \geq 44$. Further, for $dp = 6$, no convergence result is available for $N \leq 100$, whereas the approximation achieves the desired accuracy by $N = 92$, allowing the results to be obtained for $N \geq 92$ using the approximation, when no convergence result is available.

6 Conclusions

This paper has provided a major characterisation of the convergence of the steady state probabilities of a parametric Generalised Stochastic Petri Net (GSPN) with a fork-join subnet. This has led to the derivation of closed form approximate solutions under certain conditions. The GSPN has application to the coordination and synchronisation of just-in-time manufacturing systems which assemble two components (C1 and C2) into a product. C1 and C2 are manufactured by different production lines with rates λ_1 and λ_2. The components are produced on demand, with customer requests arriving at a rate λ_0. The system is controlled to ensure that one production line does not get ahead of another by more than N components.

TimeNET was used to obtain the steady state probabilities of the system for an extensive set of values of the rates, for N up to 100. Rapid convergence occurs when the three rates are not too close to each other. We defined N_{cdp} to be the

Table 6. Comparison of N_{cdp} and N_{adp} for $1 \leq dp \leq 10$

$\frac{\lambda_0}{\lambda_1}$	$\frac{\lambda_2}{\lambda_1}$	(N_{cdp}, N_{adp}) for									
		$dp=1$	$dp=2$	$dp=3$	$dp=4$	$dp=5$	$dp=6$	$dp=7$	$dp=8$	$dp=9$	$dp=10$
1.1	1.1	(6,6)	(20,19)	(42,40)	(66,63)	(90,87)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.1	1.3	(7,7)	(25,21)	(48,44)	(72,67)	(96,92)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.1	1.5	(8,7)	(27,22)	(50,44)	(74,68)	(98,92)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.1	2	(9,7)	(28,23)	(52,46)	(76,70)	(-,94)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.1	5	(10,8)	(30,27)	(54,51)	(78,75)	(-,99)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.1	10	(10,9)	(30,29)	(54,52)	(78,76)	(-,100)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.1	100	(10,10)	(30,29)	(54,53)	(78,77)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
1.3	1.1	(4,4)	(13,11)	(34,23)	(58,45)	(82,70)	(99,94)	(-,-)	(-,-)	(-,-)	(-,-)
1.3	1.3	(5,5)	(12,11)	(20,19)	(29,27)	(38,36)	(46,45)	(55,54)	(64,62)	(73,71)	(81,80)
1.3	1.5	(5,5)	(12,11)	(21,19)	(30,28)	(39,37)	(47,45)	(56,54)	(65,63)	(74,72)	(82,81)
1.3	2	(5,4)	(13,11)	(22,19)	(31,28)	(40,37)	(48,46)	(57,54)	(66,63)	(75,72)	(83,81)
1.3	5	(6,5)	(14,12)	(23,21)	(32,30)	(40,39)	(49,47)	(58,56)	(67,65)	(75,74)	(84,82)
1.3	10	(6,5)	(14,13)	(23,21)	(32,30)	(40,39)	(49,48)	(58,57)	(67,65)	(75,74)	(84,83)
1.3	100	(6,5)	(14,13)	(23,22)	(32,31)	(40,39)	(49,48)	(58,57)	(67,66)	(76,74)	(84,83)
1.5	1.1	(4,4)	(13,11)	(36,22)	(61,44)	(85,68)	(-,92)	(-,-)	(-,-)	(-,-)	(-,-)
1.5	1.3	(4,4)	(9,8)	(17,13)	(26,20)	(34,29)	(43,38)	(52,46)	(61,55)	(69,64)	(78,73)
1.5	1.5	(4,4)	(9,8)	(14,13)	(20,19)	(26,24)	(31,30)	(37,36)	(43,41)	(48,47)	(54,53)
1.5	2	(4,4)	(9,8)	(15,13)	(21,19)	(26,25)	(32,30)	(38,36)	(43,42)	(49,47)	(55,53)
1.5	5	(4,3)	(10,8)	(15,14)	(21,20)	(27,25)	(32,31)	(38,37)	(44,42)	(49,48)	(55,54)
1.5	10	(4,3)	(10,9)	(16,14)	(21,20)	(27,26)	(33,31)	(38,37)	(44,43)	(50,48)	(55,54)
1.5	100	(5,4)	(10,9)	(16,15)	(21,20)	(27,26)	(33,32)	(38,37)	(44,43)	(50,49)	(55,54)
2	1.1	(4,3)	(13,10)	(38,25)	(62,49)	(86,73)	(-,98)	(-,-)	(-,-)	(-,-)	(-,-)
2	1.3	(3,3)	(8,6)	(17,13)	(26,21)	(35,30)	(44,39)	(53,48)	(61,57)	(70,65)	(79,74)
2	1.5	(3,3)	(6,5)	(12,9)	(18,15)	(24,20)	(29,26)	(35,32)	(41,37)	(46,43)	(52,49)
2	2	(3,3)	(6,5)	(9,8)	(12,12)	(16,15)	(19,18)	(22,21)	(26,25)	(29,28)	(32,31)
2	5	(3,2)	(6,5)	(9,8)	(13,11)	(16,15)	(19,18)	(23,21)	(26,25)	(29,28)	(33,31)
2	10	(3,2)	(6,5)	(9,8)	(13,12)	(16,15)	(19,18)	(23,22)	(26,25)	(29,28)	(33,32)
2	100	(3,2)	(6,5)	(9,8)	(13,12)	(16,15)	(19,18)	(23,22)	(26,25)	(29,28)	(33,32)
5	1.1	(5,2)	(15,11)	(34,31)	(58,55)	(83,79)	(-,-)	(-,-)	(-,-)	(-,-)	(-,-)
5	1.3	(3,2)	(7,7)	(16,15)	(25,24)	(34,33)	(43,41)	(51,50)	(60,59)	(69,68)	(78,77)
5	1.5	(3,2)	(6,5)	(12,11)	(17,16)	(23,22)	(29,28)	(34,33)	(40,39)	(46,45)	(51,50)
5	2	(2,2)	(4,4)	(7,7)	(11,10)	(14,14)	(17,17)	(21,20)	(24,24)	(27,27)	(31,30)
5	5	(1,1)	(3,2)	(4,4)	(5,5)	(7,6)	(8,8)	(10,9)	(11,11)	(13,12)	(14,14)
5	10	(1,1)	(3,2)	(4,3)	(5,5)	(7,6)	(8,8)	(10,9)	(11,11)	(13,12)	(14,13)
5	100	(1,1)	(3,2)	(4,3)	(6,5)	(7,6)	(8,7)	(10,9)	(11,10)	(13,12)	(14,13)
10	1.1	(5,2)	(15,8)	(28,28)	(53,52)	(77,76)	(-,100)	(-,-)	(-,-)	(-,-)	(-,-)
10	1.3	(3,1)	(7,6)	(14,14)	(23,23)	(32,31)	(41,40)	(49,49)	(58,58)	(67,66)	(76,75)
10	1.5	(3,1)	(5,5)	(10,10)	(16,16)	(22,21)	(27,27)	(33,33)	(39,38)	(44,44)	(50,50)
10	2	(2,1)	(3,3)	(7,7)	(10,10)	(13,13)	(17,17)	(20,20)	(23,23)	(27,26)	(30,30)
10	5	(1,1)	(2,2)	(3,3)	(5,5)	(6,6)	(8,8)	(9,9)	(11,10)	(12,12)	(13,13)
10	10	(1,1)	(2,2)	(3,3)	(4,3)	(5,4)	(6,5)	(7,6)	(8,7)	(9,8)	(10,9)
10	100	(1,1)	(2,1)	(3,2)	(4,3)	(5,4)	(6,5)	(7,6)	(8,7)	(9,8)	(10,9)
100	1.1	(5,1)	(15,2)	(27,10)	(39,31)	(55,55)	(79,79)	(-,-)	(-,-)	(-,-)	(-,-)
100	1.3	(3,1)	(7,2)	(12,6)	(16,15)	(24,24)	(33,33)	(41,41)	(50,50)	(59,59)	(68,68)
100	1.5	(3,1)	(5,1)	(8,5)	(11,11)	(16,16)	(22,22)	(28,28)	(34,33)	(39,39)	(45,45)
100	2	(2,1)	(3,1)	(5,4)	(7,7)	(10,10)	(14,14)	(17,17)	(20,20)	(24,24)	(27,27)
100	5	(1,1)	(2,1)	(2,2)	(4,4)	(5,5)	(6,6)	(8,8)	(9,9)	(11,11)	(12,12)
100	10	(1,1)	(1,1)	(2,2)	(3,3)	(4,4)	(5,5)	(6,6)	(7,7)	(8,8)	(9,9)
100	100	(1,1)	(1,1)	(1,1)	(2,2)	(2,2)	(3,3)	(3,3)	(4,3)	(4,4)	(5,4)

value of N by which all the steady state probabilities are within 0.5×10^{-dp} of their converged value. This allows approximate results accurate to dp decimal places to be obtained for a GSPN with arbitrarily large $N > N_{cdp}$ from the probabilities of $GSPN_{N_{cdp}}$.

For convergence to be useful, it must occur by a value of N_{cdp} that is small enough to be calculated. Our characterisation has shown that for $dp = 3$ and $\lambda_1 < \lambda_2$ useful convergence occurs when either $\frac{\lambda_0}{\lambda_1}$ is less than about 0.9, or both $\frac{\lambda_0}{\lambda_1}$ and $\frac{\lambda_2}{\lambda_1}$ are greater than about 1.1. Thus the probabilities can be calculated for arbitrary N to a good level of accuracy for the vast majority of rates. The paper also demonstrates that N_{cdp} increases essentially linearly with dp for a wide range of rates.

For situations in which $\lambda_0 > \min(\lambda_1, \lambda_2), \lambda_1 \neq \lambda_2$, we identified two geometric progressions in the values of the converged steady state probabilities. For example, when $\lambda_0 > \lambda_1$ and $\lambda_2 > \lambda_1$ the common ratios are: $\frac{\lambda_1}{\lambda_2}$ when moving vertically from a state along the bottom edge of the CTMC to a state along the top edge; and $\frac{\lambda_1}{\lambda_0}$ when moving diagonally to the initial state from the lower right state. These progressions allowed us to derive closed form approximations for the steady state probabilities for each state in the CTMC. The approximations produce excellent results and are as good as or better than the convergence results once $N \geq N_{cdp}$. This allows results to be obtained that are not possible with tools like TimeNET, due to their computational limits.

A challenge for the future is to determine N_{cdp} for configurations satisfying $\lambda_0 > \min(\lambda_1, \lambda_2), \lambda_1 \neq \lambda_2$, without relying on tools like TimeNET. This would make it possible to obtain the steady state probabilities of all states of $CTMC_N$ for arbitrary $N \geq N_{cdp}$ directly from equations (4) and (5). This would also extend our results into the region where tools cannot obtain N_{cdp}. Another challenging task is to determine if similar approximations can be devised for configurations in which $\lambda_0 \leq \min(\lambda_1, \lambda_2)$, and when $\lambda_1 = \lambda_2$. Finally a very significant generalisation is to consider 3 or more branches. Our early results show similar convergence trends, but they need to be fully characterised.

Acknowledgements. The authors gratefully acknowledge the constructive comments of the reviewers.

References

1. Baccelli, F., Massey, W.A., Towsley, D.: Acyclic fork-join queuing networks. Journal of the ACM 36(3), 615–642 (1989)
2. Flatto, L., Hahn, S.: Two Parallel Queues Created by Arrivals with Two Demands I. SIAM Journal of Applied Mathematics 44, 1041–1053 (1984)
3. Flatto, L.: Two Parallel Queues Created by Arrivals with Two Demands II. SIAM Journal of Applied Mathematics 45, 861–878 (1985)
4. Nelson, R., Towsley, D., Tantawi, A.N.: Performance Analysis of Parallel Processing Systems. IEEE Transactions on Software Engineering 14(4), 532–540 (1988)
5. Nelson, R., Tantawi, A.N.: Approximate Analysis of Fork/Join Synchronization in Parallel Queues. IEEE Transactions on Computers 37(6), 739–743 (1988)

6. Kim, C., Agrawala, A.K.: Analysis of the Fork-Join Queue. IEEE Transactions on Computers 38(2), 250–255 (1989)
7. Liu, Y.C., Perros, H.G.: A Decomposition Procedure for the Analysis of a Closed Fork/Join Queueing System. IEEE Transactions on Computers 40(3), 365–370 (1991)
8. Lui, J.C.S., Muntz, R.R., Towsley, D.: Computing Performance Bounds of Fork-Join Parallel Programs under a Multiprocessing Environment. IEEE Transactions on Parallel and Distributed Systems 9(3), 295–311 (1998)
9. Makowski, A., Varma, S.: Interpolation Approximations for Symmetric Fork-Join Queues. Performance Evaluation 20, 145–165 (1994)
10. Varki, E.: Mean value technique for closed fork-join networks. In: SIGMETRICS 1999: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 103–112 (1999)
11. Varki, E.: Response Time Analysis of Parallel Computer and Storage Systems. IEEE Transactions on Parallel and Distributed Systems 12(11), 1146–1161 (2001)
12. Harrison, P., Zertal, S.: Queueing models of RAID systems with maxima of waiting times. Performance Evaluation 64, 664–689 (2007)
13. Song, J.S., Zipkin, P.: Supply Chain Operations: Assemble-to-Order Systems. In: Handbook in OR&MS. ch. 11, vol. 11, pp. 561–596. Elsevier, Amsterdam (2003)
14. Parry, G., Graves, A. (eds.): Build to Order: The Road to the 5-Day Car. Springer, London (2008)
15. Bause, F., Kritzinger, P.S.: Stochastic Petri Nets - An Introduction to the Theory, 2nd edn. Vieweg (2002)
16. Ajmone Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with Generalized Stochastic Petri Nets. John Wiley and Sons, Chichester (1995)
17. Lilith, N., Billington, J., Freiheit, J.: Approximate Closed-Form Aggregation of a Fork-Join Structure in Generalised Stochastic Petri Nets. In: Proc. 1st Int. Conference on Performance Evaluation Methodologies and Tools, Pisa, Italy, International Conference Proceedings Series, vol. 180, Article 32, 10 pages. ACM Press, New York (2006)
18. Doğançay, K.: An Asymptotic Convergence Result for the Aggregation of Closed Fork-Join Generalised Stochastic Petri Nets. In: Proc. IEEE Region 10 Conference, TENCON 2009, Singapore, November 23-26, 6 pages (2009)
19. Billington, J., Gallasch, G.E.: Steady State Markov Analysis of a Controlled Fork-Join Network. Technical Report CSEC-41, Computer Systems Engineering Centre, University of South Australia (June 30, revised July 15, 2010)
20. Gallasch, G.E., Billington, J.: A Study of the Convergence of Steady State Probabilities in a Closed Fork-Join Network. In: Proc. 8th International Symposium on Automated Technology for Verification and Analysis (ATVA 2010), Singapore, September 21-24. LNCS, vol. 6252, pp. 143–157. Springer, Heidelberg (2010)
21. University of Ilmenau: TimeNET, http://www.tu-ilmenau.de/TimeNET

Reasoning about Safety and Progress Using Contracts

Imene Ben-Hafaiedh, Susanne Graf, and Sophie Quinton

Université Joseph Fourier, VERIMAG

Abstract. Designing concurrent or distributed systems with complex architectures while preserving a set of high-level requirements through all design steps is not a trivial task. Building upon a generic notion of *contract framework* which relies on a *component framework* and two refinement relations: *conformance* and *refinement under context*, we provide a condition under which circular reasoning can be used for checking *dominance*, i.e. refinement between contracts. We then propose an instantiation of such a contract framework for safety and progress requirements in component systems with data exchange. This allows us to prove non-trivial properties of a protocol for tree-like networks.

1 Introduction

We aim at a scalable methodology for design and verification of distributed component systems of arbitrary size with complex architectures which preserves a set of high-level requirements through all design steps. Like in contract-based design [1], we use contracts to constrain, reuse and replace implementations.

In this paper we formalize and extend the verification methodology introduced in [2] to distributed component systems of arbitrary size and we show its usefulness for proving safety and progress properties in networked systems. This methodology consists in two phases: (1) define a general notion of contract framework stating the necessary ingredients — a component framework, notions of conformance (for ensuring global properties φ), satisfaction (of contracts by implementations), and dominance (refinement between contracts). Rules for establishing dominance and validity conditions for them are provided. (2) for any particular application, one only has to define instantiations of these generic notions and check the validity conditions. Once the concrete framework has been defined, the rules for dominance can be applied without any further proofs.

For expressing the rich, yet abstract specifications required by our example, we propose a formalism similar to symbolic transition systems as introduced in [3], which we extend in several ways. We define progress constraints generalizing the usual strong and weak fairness and we decorate control states with invariants on state variables. We also consider an explicit composition model represented by sets of *connectors*. Each connector defines a set of interactions and a transformation on (non persistent) port variables, where *ports* name transitions of the local components involved in the interaction. For achieving scalability, we base

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 436–451, 2010.
© Springer-Verlag Berlin Heidelberg 2010

verification on an abstract semantics in which explicit values of state variables are abstracted by the defined state invariants. Given the complexity of the specifications, not having to prove the correctness of the proof rules in this concrete setting is very helpful.

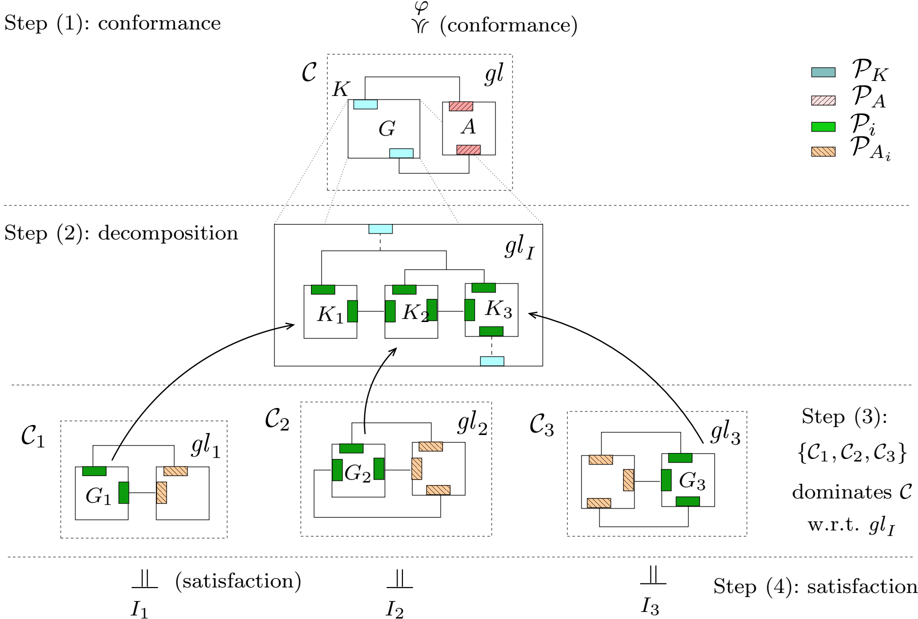


Fig. 1. Methodology steps ensuring that $gl\{A, gl_I\{I_1, I_2, I_3\}\} \preccurlyeq \varphi$

Methodology. Figure 1 illustrates our design and verification methodology. It is represented in a top-down fashion in which high-level properties are pushed progressively from the overall system into atomic components — which we call implementations. As usual, this is just a convenient representation; in real life, we will always achieve the final picture in several iterations alternatively going up and down. We are interested in systems with a complex architecture which are potentially of arbitrary size.

We suppose given a global property φ which the system K under construction has to realize together with an environment on which we may have some knowledge, expressed by a property A. φ and A are expressed w.r.t the interface $\mathcal{P}_K$ of K. We proceed as follows: (1) define a *contract* $\mathcal{C}$ for $\mathcal{P}_K$ which *conforms* to φ; (2) define K as a composition of subcomponents K_i and a contract $\mathcal{C}_i$ for each of them; possibly iterate this step if needed. (3) prove that any set of implementations (components) for K_i *satisfying* the contracts $\mathcal{C}_i$, when composed, satisfies the top-level contract $\mathcal{C}$ (*dominance*) — and thus guarantees φ; (4) provide such implementations.

The global property φ appears at the top of Figure 1, while the implementations I_i are at the bottom.

The correctness proof for a particular system is split into 3 phases: *conformance* of the top-level contract $\mathcal{C}$ to φ, *dominance* between the contracts $\mathcal{C}_i$ and $\mathcal{C}$, *satisfaction* of the $\mathcal{C}_i$ by the implementation I_i.

To be more precise, we use the notion of *context* for an interface $\mathcal{P}$ to describe how a component with interface $\mathcal{P}$ is intended to be connected to its environment and provides a property A expected from this environment. In the sequel, we denote composition operators by gl — standing for *"glue"* [4]. A context is then of the form (gl, A). A *contract* for an interface $\mathcal{P}$ consists of a context (gl, A) and a property G on $\mathcal{P}$ that the component under design must ensure in the given context in order to *satisfy* this contract. *Conformance* relates properties of closed systems and *dominance* relates contracts.

Related work. We propose here 3 improvements with respect to [2]: (a) we do not suppose a fixed composition operator: we encompass any composition satisfying some basic properties; (b) we extend the definition of contract framework to take into account port hiding which is a key ingredient for proving refinement between specifications at different levels of granularity; (c) we provide a complex application using an instantiation with variables and data transfer and allowing expression of liveness properties. The proof steps are performed automatically.

Interfaces [5] have been proposed for a purpose similar to ours. However, we are interested here in rich exogenous composition operators which allow to represent abstractions of protocols, middleware components and orchestrations whereas assumptions and guarantees should constrain peers at the same or at an upper layer. These composition operators cannot be encoded into interface automata, which are I/O based.

Other formalisms for describing such rich connectors abstractly have been proposed, e.g., the Kell calculus [6] or the connector calculus Reo [7]. Kell is, however, mainly concerned with obtaining correctly typed connectors, and Reo supposes independence amongst connectors and does not take into account constraints imposed by components. The composition operators used in our application are defined using a subset of the rich connectors of the BIP component framework [8] because these connectors have the required expressiveness, define interaction with component behaviors and handle conflicting connectors.

Organization. Section 2 introduces and extends the notions from [2] of contract framework and properties that *conformance*, *dominance* and *satisfaction* must ensure in order to support this methodology. In section 3, we give an instantiation of this framework based on symbolic transition systems and rich connectors, which is expressive enough for the safety and progress properties we want to prove. Finally, Section 4 applies the methodology to a resource sharing algorithm in a networked system of arbitrary size: the actual conformance, dominance and satisfaction proofs are automated in a tool developed for this purpose.

2 A Contract-Based Design Framework and Methodology

We develop our methodology on a generic framework that supports hierarchical components and mechanisms to reason about composition. The following notions

and properties form the basis of this framework. Here, we use glue operators [4] to generalize the operation of parallel composition found in most traditional frameworks. The notion of component is intentionally kept very abstract to encompass various frameworks. It can be e.g. a labeled transition system, but it can also have a structural part, e.g. it can be a BIP component.

Definition 1 (Component framework). *A* component framework *is a structure of the form* $(\mathcal{K}, GL, \circ, \cong)$ *where:*

- $\mathcal{K}$ *is a set of* components. *Each component* $K \in \mathcal{K}$ *has as its* interface *a set of* ports, *denoted* $\mathcal{P}_K$.
- GL *is a set of* glue (composition) operators. *A glue is a partial function* $2^{\mathcal{K}} \longrightarrow \mathcal{K}$ *transforming a set of components into a new component. Each* $gl \in GL$ *is associated with a set of ports* S_{gl} *from the original set of components — called its* support set *— and a new interface* $\mathcal{P}_{gl}$ *for the new component — called its* exported interface. *A composition* $K = gl(\{K_1, \ldots, K_n\})$ *is defined if* $K_1, \ldots, K_n \in \mathcal{K}$ *have disjoint interfaces,* $S_{gl} = \bigcup_{i=1}^{n} \mathcal{P}_{K_i}$ *and the interface of* K *is* $\mathcal{P}_{gl}$, *the exported interface of* gl.
- $\cong \subseteq \mathcal{K} \times \mathcal{K}$ *is an equivalence relation. In general, this equivalence is derived from equality or equivalence of semantic sets.*
- $\circ$ *is a partial operation on* GL *to hierarchically compose glues.* $gl \circ gl'$ *is defined if* $\mathcal{P}_{gl'} \subseteq S_{gl}$. *Then, its support set is* $S_{gl} \backslash \mathcal{P}_{gl'} \cup S_{gl'}$ *and its interface is* $\mathcal{P}_{gl}$ *(cf. Figure 2). Furthermore,* $\circ$ *must be coherent with* $\cong$ *in the sense that* $gl\{gl'\{\mathcal{K}_1\}, \mathcal{K}_2\} \cong (gl \circ gl')\{\mathcal{K}_1 \cup \mathcal{K}_2\}$ *for any sets of components* $\mathcal{K}_i$ *such that all terms are defined.*

To simplify the notation, we write $gl\{K_1, \ldots, K_n\}$ instead of $gl(\{K_1, \ldots, K_n\})$. Figure 2 shows how hierarchical components and connectors are built from atomic ones. Note that exported ports of internal connectors (which are not connected) are not represented in this figure.

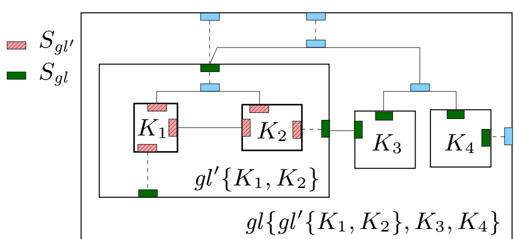


Fig. 2. Hierarchical components and connectors

We use the notion of *context* to restrict how a component may be further composed. The set of contexts is denoted Γ.

Definition 2 (Context). *A* context *for an interface* $\mathcal{P}$ *is a pair* (E, gl) *where* $E \in \mathcal{K}$ *is such that* $\mathcal{P} \cap \mathcal{P}_E = \emptyset$ *and* $gl \in GL$ *is defined on* $\mathcal{P} \cup \mathcal{P}_E$.

We introduce two refinement relations to reason about contracts: *conformance*, which we informally introduced when discussing our methodology; and *refinement under context*, used to define *satisfaction* and *dominance*. Refinement under context is usually considered as a derived relation and chosen as the weakest relation implying conformance and ensuring *compositionality*, i.e., preservation by composition. We loosen the coupling between these two refinements to obtain stronger reasoning schemata for dominance.

Definition 3 (Contract framework). *A* contract framework *is a tuple* $(\mathcal{K}, GL, \circ, \cong, \{\sqsubseteq_c\}_{c \in \Gamma}, \preccurlyeq)$ *where:*

- $(\mathcal{K}, GL, \circ, \cong)$ *is a component framework.*
- $\{\sqsubseteq_c\}_{c \in \Gamma}$ *is a set of* refinement under context *relations, one for each context in* Γ*. Given a context* (E, gl) *for an interface* $\mathcal{P}$*,* $\sqsubseteq_{E,gl}$ *is a preorder over the set of components on* $\mathcal{P}$*.*
- $\preccurlyeq \subseteq \mathcal{K} \times \mathcal{K}$ *is a* conformance *relation between components with the same interface. It is a preorder such that for any* K_1*,* K_2 *on the same interface* $\mathcal{P}$ *and for any context* (E, gl) *for* $\mathcal{P}$*,* $K_1 \sqsubseteq_{E,gl} K_2 \implies gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$*.*

Example 1. Typical notions of conformance $\preccurlyeq$ are *trace inclusion* and *simulation.*

For these notions of conformance, refinement under context (denoted $\sqsubseteq^{\preccurlyeq}$) is usually defined as the weakest preorder included in $\preccurlyeq$ that is compositional:

$$K_1 \sqsubseteq^{\preccurlyeq}_{E,gl} K_2 \triangleq gl\{K_1, E\} \preccurlyeq gl\{K_2, E\}$$

Note that there are cases where a stronger notion of refinement under context allows more powerful reasoning, e.g. circular reasoning as used later in this paper.

Definition 4 (Contract). *A* contract $\mathcal{C}$ *for an interface* $\mathcal{P}$ *consists of:*

- *a context* $\mathcal{E} = (A, gl)$ *for* $\mathcal{P}$*;* A *is called the* assumption
- *a component* G *on* $\mathcal{P}$ *called the* guarantee

We write $\mathcal{C} = (A, gl, G)$ rather than $\mathcal{C} = ((A, gl), G)$. The interface of the environment is implicitly defined by gl while A expresses a constraint on it and G a constraint on the refinements of K. The "mirror" contract $\mathcal{C}^{-1}$ of $\mathcal{C}$ is (G, gl, A), i.e. a contract for the environment.

Definition 5 (Satisfaction of contract). *A component* K satisfies *a contract* $\mathcal{C} = (A, gl, G)$*, denoted* $K \models \mathcal{C}$*, if and only if* $K \sqsubseteq_{A,gl} G$*.*

In interface theories [5], a single automaton is used to represent both A and G (gl is predefined), namely $gl\{A, G\}$. Only one pair (A, G) corresponds to an interface, because each transition is controlled either by the component or the environment. However, in frameworks with rendez-vous interaction, several pairs (A, G) can correspond to the same interface, as both the component and its environment may prevent a rendez-vous from taking place. This is why we keep assumptions and guarantees separate.

Our notion of contract has a structural part, which makes this definition very general by encompassing any composition framework. A more practical advantage is related to system design: it allows us to separate the architecture and

the requirements of the system under construction, which evolve independently during the development process. In particular, in frameworks where interaction is rich, refinement can be ensured by relying heavily on the structure of the system and less importantly on the behavioral properties of the environment.

Dominance is the key notion that distinguishes reasoning in a contract or interface framework from theories based on refinement between components. Contract $\mathcal{C}$ is said to dominate contract $\mathcal{C}'$ if every implementation of $\mathcal{C}$ — i.e., every component satisfying $\mathcal{C}$ — is also an implementation of $\mathcal{C}'$. Intuitively, this is achieved by a $\mathcal{C}'$ that has a stronger promise or a weaker assumption than $\mathcal{C}$.

In our general setting — which does not refer to any particular composition or component model — it is not sufficient to define dominance just on a pair of contracts. A typical situation that we have to handle is that of a hierarchical component depicted in Figure 1, where a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ is defined for the inner components (on disjoint interfaces $\{\mathcal{P}_i\}_{i=1}^n$) and a contract $\mathcal{C}$ for the hierarchical component whose interface is the exported interface of a composition operator gl_I defined on $P = \bigcup_{i=1}^n \mathcal{P}_i$. It looks attractive to solve such a problem by defining a contract algebra as in [9], as checking dominance boils then down to checking whether $\widetilde{gl}\{\mathcal{C}_1, \dots, \mathcal{C}_n\}$ dominates $\mathcal{C}$ for some operator $\widetilde{gl}$ on contracts. This is, however, not possible for arbitrary component frameworks. We thus provide a broader dominance defined directly for a set of contracts $\{\mathcal{C}_i\}_{i=1}^n$ and a contract $\mathcal{C}$ to be dominated w.r.t a composition operator gl_I.

In order to allow hiding ports of the lower-level contracts which do not appear at the interface of the top-level contract, we relax the constraints on the composition operators by only requiring that they agree on their common ports. For this, we need a notion of *projection* of a component K onto a subset P' of its interface, which defines a component denoted $\Pi_{P'}(K)$ with interface P'. This notion is quite natural and must preserve some properties detailed in [10]. Hence the following semantic definition of dominance (notations are those of Figure 1).

Definition 6 (Dominance). *$\{\mathcal{C}_i\}_{i=1}^n$ dominates $\mathcal{C}$ w.r.t. gl_I iff:*

– *for every i, there exists a glue gl_{E_i} s.t. $gl \circ gl_I = gl_i \circ gl_{E_i}$*
– *for any components $\{K_i\}_{i=1}^n$, $(\forall i, K_i \models \mathcal{C}_i) \implies \Pi_P(gl_I\{K_1, \dots, K_n\}) \models \mathcal{C}$*

We present a generalization of the sufficient condition for dominance proposed in [2] that handles port hiding. The proof is similar to that of [2] and requires a specific property called soundness of circular reasoning. Circular reasoning is sound if for any K, G, A, E, gl such that the terms are defined, the following holds: $K \sqsubseteq_{A,gl} G \wedge E \sqsubseteq_{G,gl} A \implies K \sqsubseteq_{E,gl} G$. More details are given in [10].

Theorem 1. *If circular reasoning is sound and $\forall i.\ \exists gl_{E_i}.\ gl \circ gl_I = gl_i \circ gl_{E_i}$, then to prove that $\{\mathcal{C}_i\}_{i=1..n}$ dominates $\mathcal{C}$ w.r.t. gl_I, it is sufficient to prove that:*

$$\begin{cases} \Pi_P(gl_I\{G_1, \dots, G_n\}) \models \mathcal{C} \\ \forall i, \Pi_{\mathcal{P}_{A_i}}(gl_{E_i}\{A, G_1, \dots, G_{i-1}, G_{i+1}, \dots, G_n\}) \models \mathcal{C}_i^{-1} \end{cases}$$

This shows that the proof of a dominance relation boils down to a set of refinement checks, one for proving refinement between the guarantees, the second for discharging individual assumptions. A proof is given in [10].

Methodology. We now extend our design and verification methodology to recursively defined systems so that we can handle systems representing component networks of arbitrary size defined by a component grammar as follows:

- a set of terminal symbols $\{A, I_1, \dots, I_k\}$ representing implementations;
- a set of nonterminal symbols $\{S, K_0, K_1, \dots, K_n\}$ representing hierarchical components; S, which defines the top-level closed system, is the axiom;
- a set of rules corresponding to design steps which define each non-terminal either as a composition of subsystems or as an implementation:
  - $S \longrightarrow gl\{A, K_0\}$.
  - For $i \in [0, n]$, at least one rule either of the form $K_i \longrightarrow I_j$ ($j \in [1, k]$) or $K_i \longrightarrow gl_{\Sigma_i}\{K_j\}_{j\in\Sigma_i}$, where Σ_i a set of indices and gl_{Σ_i} a composition operator on the union of the interfaces of the K_j.

Unlike classical network grammars, we use "rich" composition operators and are not limited to flat regular networks, as for example in [11]. We now instantiate the methodology of Figure 1 for such component networks. The same four steps are presented, namely conformance, decomposition, dominance and satisfaction.

1. formulate a top-level requirement φ characterizing the closed system defined by the system and its environment, define a contract $\mathcal{C} = (A, gl, G)$ associated with K_0 and prove that $gl\{A, G\} \preccurlyeq \varphi$
2. define for every non terminal K_i a contract $\mathcal{C}_{K_i} = (A_{K_i}, gl_{K_i}, G_{K_i})$ such that for every rule $K_l \longrightarrow gl_{\Sigma_l}\{K_j\}_{j\in\Sigma_l}$ having an occurrence of K_i on the right hand side, there exists gl_{E_i} such that $gl_{K_l} \circ gl_{\Sigma_l} = gl_{K_i} \circ gl_{E_i}$
3. for each $K_i \longrightarrow gl_{\Sigma_i}\{K_j\}_{j\in\Sigma_i}$, show that $\{\mathcal{C}_{K_j}\}_{j\in\Sigma_i}$ dominates $\mathcal{C}_{K_i}$ w.r.t gl_{Σ_i}
4. prove that implementations satisfy their contract: $K_i \longrightarrow I_j \implies I_j \models \mathcal{C}_{K_i}$

Theorem 2. *Let $\mathcal{G}$ be a grammar such that all methodology steps have been completed to guarantee a requirement φ. Any component system corresponding to a word accepted by $\mathcal{G}$ satisfies φ.*

The proof is a simple induction on the number of steps required for deriving the component from S, showing that conformance is preserved from the left-hand side to the right-hand side of a rule. If φ can express progress and if dominance preserves progress, this methodology is sufficient for systems with a unique requirement but also for multiple requirements decomposed according to the same network grammar.

3 A Contract Framework with Data for Safety and Progress

In this section, we define a contract framework in order to prove safety and progress properties of distributed systems. We choose to use composition operators based on the BIP interaction model [4,12] because of their expressiveness and their properties making them suitable for structural verification. Our framework handles variables, guards and data transfer — which are supported by the BIP interaction model [13] — and furthermore is adequate for loose specifications.

Components. A component is defined by a labeled transition system enriched with variables. In order to allow abstract descriptions of components, we handle predicates on variables rather than concrete values. Except for τ, which denotes internal actions, labels are ports of the component interface. For example, a transition t labeled by port p denotes that the component will perform the action associated with t only if an interaction in which p is involved happens. Our components also provide some progress properties which are described below.

A port p is sometimes represented along with its associated variables $x_1, \ldots, x_n$, which is denoted $p[x_1, \ldots, x_n]$. Without loss of generality, we suppose in the following that a port is associated with exactly one variable. We suppose given a set of predicates that is closed by $\wedge$ and $\vee$.

Definition 7 (Component). *A* component *is a tuple* $(TS, X, Inv, g, f, Prog)$*:*

- $TS = (Q, q^0, P \cup \{\tau\}, \longrightarrow)$ *is a labeled transition system: Q is a set of* states*, $q^0 \in Q$ is the* initial state*, $P \cup \{\tau\}$ is a set of labels. $\longrightarrow \subseteq Q \times P \cup \{\tau\} \times Q$ is a* transition *relation. Elements of P are* ports *and τ labels* internal *transitions. As usual, a transition $(q, p, q') \in \longrightarrow$ is denoted $q \xrightarrow{p} q'$;*
- *X is a set of* variables*. Some variables are associated with a (unique) port; $X^{st} \subseteq X$ contains* state variables *which are denoted $st_1, \ldots, st_s$. Relation R relates*[1] *variables in X to variables in X^{st};*
- *Inv associates with every $q \in Q$ a* state invariant *Inv_q that is a predicate on X^{st};*
- *g associates with every transition t a* guard *g_t, i.e. a predicate on X^{st};*
- *f associates with every transition t an* action *f_t defined as a predicate on $X^{st} \cup \{x_\gamma\} \cup X^{st}_{new}$ where x_γ is the variable associated with the port labeling t*[2] *and $X^{st}_{new} = \{st_1^{new}, \ldots, st_s^{new}\}$ represents the "updated" variables;*
- *$Prog$ a set of* progress properties *(see below).*

Progress properties. When considering abstract specifications, progress properties are useful to exclude behaviors staying forever in some particular states or loops. We adapt usual weak and strong fairness conditions to component systems: a *progress property $pr \in Prog$* for a component K is a pair of transition sets (T_c, T_p), where T_c is called the *condition* and T_p the *promise*.

We define the set of *progress states* of T_p, denoted $start(T_p)$, as the set of initial states of transitions of T_p.

(T_c, T_p) is a valid progress property iff: considering an execution σ of K in some context containing infinitely many T_c-transitions, in every state of $start(T_p)$ occurring infinitely often, at least one transition of T_p appears infinitely often in σ, unless the environment forbids it. $(\top, T_p)$ denotes *unconditional progress*, which means that σ cannot stay forever in $start(T_p)$ without firing infinitely often a transition of T_p.

Note that (T_c, T_p) is trivially satisfied if no T_c-transition can be fired infinitely often. When T_p is empty or not reachable from any "T_c-loop", (T_c, T_p) is

[1] Non-state variables are transient. R produces their value whenever it is necessary.

[2] If there is no associated variable (t is labeled by p_γ with $\gamma \in \mathcal{I}_{obs}$ or by τ), f_t is a predicate on $X^{st} \cup X^{st}_{new}$.

a progress property only if no T_c-transition can be fired infinitely often. Monotonicity properties w.r.t. progress which allow inferring new progress properties from existing ones are given in [10].

Semantics. The concrete semantics of a component is the usual SOS semantics for labeled transition systems. We do not need it in the following because we only work with an abstract semantics of components: the latter is a labeled transition system in which there exists a transition iff there exists a concrete valuation of the variables for which the transition can be fired. Our semantics is a *closed* semantics, because we suppose that the environment of the component does not affect the values of the variables attached to ports labeling transitions. This strongly motivates a design approach based on contracts, that is, on closed systems.

Definition 8 (Abstract semantics). *Let* $K = (TS, X, g, f, Inv, Prog)$ *be a component. The abstract semantics of* K *is the transition system* $(Q, q^0, P, \hookrightarrow)$ *where* $q \overset{p[x]}{\hookrightarrow} q'$ *iff there exist a transition* $t = (q \xrightarrow{p[x]} q')$ *such that the predicate* $(st_1, \dots, st_s)\, R\, x \wedge Sem_t$ *is satisfiable, where* Sem_t *denotes* $Inv_q \wedge g_t \wedge f_t \wedge Inv_{q'}$*.*

Note that a transition $t = (q \xrightarrow{p} q')$ is not preserved in the semantics if f_t is not consistent with $Inv_{q'}$ — meaning that firing t leads to a state in which $Inv_{q'}$ cannot hold. Thus, in order to avoid deadlocks, the state invariants must respect some consistency and completeness conditions.

Composition. We now define the composition operators that allow us to build complex components based on atomic ones. These composition operators are called *interaction models* and they are made of *connectors*.

From the possible synchronizations offered by the BIP framework (see [12]), we keep only two basic types of connectors: *rendez-vous* connectors require *all* ports to be activated in order for the interaction to take place and involve data transfers; interactions in an *observation* connector can take place as soon as *any* port is activated, and no data is exchanged. Adding observation connectors does not modify the set of interactions which can be fired in a given state, so this does not change the behavior of the system, hence their name. Two (or more) connectors of the same type can be composed to build a *hierarchical* connector simply by using the exported port of one connector as an element of the support set of the other.

Definition 9 (Rendez-vous connector). *A* rendez-vous *connector* $\gamma = (p[x], P, \delta)$ *is defined by:*

- $p[x]$*, the* exported port *and* $P = \{p_1[x_1], \dots, p_k[x_k]\}$*, the* support set *of ports*
- $\delta = (G, \mathcal{U}, \mathcal{D})$ *where:*
  - G *is the* guard*, that is, a predicate on* $X = \{x_1, \dots, x_k\}$
  - $\mathcal{U}$ *is the* upward update function *defined as a predicate on* $X \cup \{x\}$
  - *For* $x_i \in X, \mathcal{D}_{x_i}$ *is a* downward update function*, i.e. a predicate on* $\{x\} \cup \{x_i\}$

where $\mathcal{D}_{x_i}$ *is the function that returns the projection of* $\mathcal{D}$ *corresponding to* x_i*.*

As *observation* connectors do not involve data transfer, they have neither guard nor $\mathcal{U}$ nor $\mathcal{D}$ predicates. The variables attached to ports are useless and thus hidden. Hence the following definition.

Definition 10 (Observation connector). *An* observation *connector* $\gamma = (p, P)$ *is defined by an* exported port p *and a* support set $P = \{p_1, \dots, p_k\}$.

To avoid cyclic connectors, we require also that $p \notin P$. Two connectors γ_1 and γ_2 are *disjoint* if $p_1 \neq p_2$, $p_1 \notin P_2$ and $p_2 \notin P_1$. Note that P_1 and P_2 may have ports in common, as a port may be connected to several connectors.

We can now define our composition operators as sets of connectors.

Definition 11 (Interaction model). *An interaction model $\mathcal{I}$ defined on a set of ports P is a set of disjoint connectors such that P is the union of the support sets of the rendez-vous connectors of $\mathcal{I}$. We denote by $\mathcal{I}_{rdv}$ the set of its rendez-vous connectors and $\mathcal{I}_{obs}$ the set of its observation connectors.*

We associate with an interaction model $\mathcal{I}$ an interface $\mathcal{P}_\mathcal{I}$ consisting of the set of the exported ports of its connectors. This means that the interface of the component resulting from a composition using $\mathcal{I}$ has only these exported ports as labels. $X_\mathcal{I}$ denotes the set of variables associated with the ports of $\mathcal{P}_\mathcal{I}$.

Merge of connectors is the operation that takes two connectors defining together a hierarchical connector and returns a connector of a basic type. Merge is defined for rendez-vous connectors in [13] (where it is called flattening). We restrict this definition so as to preserve associativity of the upward and downward functions. Merge of observation connectors has been described in [12]. These definitions extend naturally to our interaction models, where rendez-vous and observation connectors are merged separately (see [10]).

We now define composition: given a set of components $K_1, \dots, K_n$ and an interaction model $\mathcal{I}$, we build a compound component denoted $\mathcal{I}\{K_1, \dots, K_n\}$, with $\mathcal{P}_\mathcal{I}$ as interface. As we do not allow sets of ports as labels of transitions, we require that connectors of $\mathcal{I}$ have at most one port of the same component in their support set. Composition is rather technical but not surprising. It does not involve hiding of ports. Besides, a variable of $\mathcal{I}\{K_1, \dots, K_n\}$ is a variable of some K_i or a variable associated with the exported port of some $p_\gamma \in \mathcal{I}$.

Definition 12 (Composition of components). *Let $\{P_i\}_{i=1}^n$ be a family of pairwise disjoint interfaces and $P = \bigcup_{i=1}^n P_i$. Let $\mathcal{I}$ be an interaction model on P. For $i \in [1, n]$, let $K_i = (TS_i, X_i, g_i, f_i, Inv_i, Prog_i)$ be a component on P_i. The* composition *of $K_1, \dots, K_n$* with *$\mathcal{I}$ is a component $(TS, X, g, f, Inv, Prog)$ such that:*

- *$TS = (Q, q^0, \mathcal{P}_\mathcal{I} \cup \{\tau\}, \longrightarrow)$ with $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \dots, q_n^0)$ and where $\longrightarrow$ is the least set of transitions satisfying the following rules[3]:*

$$\frac{(p_\gamma, P_\gamma, \delta_\gamma) \in \mathcal{I}_{rdv} \quad \forall i \in [1, n].\ q_i \xrightarrow{P_i \cap P_\gamma}_i q_i'}{(q_1, \dots, q_n) \xrightarrow{p_\gamma} (q_1', \dots, q_n')} \qquad \frac{\exists i \in [1, n].\ q_i \xrightarrow{\tau}_i q_i'}{(q_1, \dots, q_n) \xrightarrow{\tau} (q_1, \dots, q_i', \dots, q_n)}$$

with the convention that $q_i \xrightarrow{\emptyset}_i q_i'$ iff $q_i = q_i'$. Note that $|P_i \cap P_\gamma| \leq 1$.

[3] The rule for connectors in $\mathcal{I}_{obs}$ is similar to the one for rendez-vous connectors except that any subset of the support set P_γ may participate in the interaction.

- $X^{st} = \bigcup_{i=1}^{n} X_i^{st}$ *and* $X = \bigcup_{i=1}^{n} X_i \cup X_\mathcal{I}$
  The relation R *between variables in* X *and state variables is defined as:*
  Case 1: $x \in X_i$ *for some* $i \in [1, n]$. $x\, R\, (st_1, \dots, st_s)$ *iff* $x\, R_i\, (st_1^i, \dots, st_{s_i}^i)$, *where* $\{st_1^i, \dots, st_{s_i}^i\} = X_i^{st} \subseteq X^{st}$.
  Case 2: $x \in X_\mathcal{I}$. *Then* x *is associated with the exported port* p_γ *of a rendezvous connector* $\gamma = (p_\gamma, P_\gamma, \delta) \in \mathcal{I}_{rdv}$. *Let* $k = |P_\gamma|$. $\mathcal{U}_\gamma$ *is a predicate on* $\{x_1, \dots, x_k\} \cup \{x\}$, *where every* x_i *is associated with a port of* P_γ. *Without loss of generality, we suppose each* x_i *is a variable of component* K_i.
  Then $x\, R\, (st_1, \dots, st_s)$ *is defined iff:*
  $$\exists v_1, \dots, v_k.\ (\forall i \in [1, k].\ v_i\, R_i\, (st_1^i, \dots, st_{s_i}^i)) \wedge \mathcal{U}_\gamma[x_1/v_1, \dots, x_k/v_k]$$
  where $\mathcal{U}_\gamma[x_1/v_1, \dots, x_k/v_k]$ *is the predicate on* x *obtained by replacing the variables* $x_1, \dots, x_k$ *by values* $v_1, \dots, v_k$ *compatible with the local relations* R_i *between the* x_i *and the local state variables.*
- *For each* $q \in Q$, $Inv_q = \bigwedge_{\{i=1\}}^{n} Inv_{q_i}$
- *Consider* $t = (q_1, \dots, q_n) \xrightarrow{p_\gamma} (q_1', \dots, q_n')$ *for* $\gamma \in \mathcal{I}_{rdv}$[4]. *W.l.o.g., we suppose* $P_\gamma = \{x_1, \dots, x_k\}$ *with* $x_i \in P_i$ *for every* i *in* $[1, k]$. *For* $i \in [1, k]$, *the local transition* $(q_i \xrightarrow{p_i[x_i]}_i q_i')$ *corresponding to* t *is denoted* $\pi_i(t)$. *Again,* $\{st_1^i, \dots, st_{s_i}^i\} = X_i^{st} \subseteq X^{st}$.
  - $g_t(st_1, \dots, st_s)$ *holds iff the following holds:*
    - $\forall i \in [1, k].\ g_{t_i}(st_1^i, \dots, st_{s_i}^i)$
    - $\exists v_1, \dots, v_k.\ (\forall i \in [1, k].\ v_i\, R_i\, (st_1^i, \dots, st_{s_i}^i)) \wedge G[x_1/v_1, \dots, x_k/v_k]$
  - $f_t(st_1, \dots, st_s, x_\gamma, st_1^{new}, \dots, st_s^{new})$ *holds iff* $\exists v_1, \dots, v_k$ *s.t. it holds that:*
    - $\mathcal{D}[x_1/v_1, \dots, x_k/v_k]$, *which is a predicate on* x_γ
    - $\forall i \in [1, k].\ f_{\pi_i(t)}[x_i/v_i]$, *which is a predicate on* $X_i^{st} \cup X_{i,new}^{st}$
- *The set Prog of progress properties is defined below (see definition 13)*

We never explicitly construct all (strongest) progress properties for a composition: compositions are only built as far as needed to prove dominance. Thus, we only give below a condition for checking that a pair of sets (T_c, T_p) is a progress property of a composition by checking that the projections of (T_c, T_p) onto individual components are local progress properties.

Definition 13 (Progress property in a composition). (T_c, T_p) *is a progress property of* $\mathcal{I}\{K_1, \dots, K_n\}$ *if* $\forall i \in [1, n]$*:*

- *either* $\pi_i(T_p)$ *never contains more than one joint transition of* $\mathcal{I}$ *from the same state and then* $(\pi_i(T_c), \pi_i(T_p))$ *is a local progress property.*
- *or it does, and then we split* $\pi_i(T_p)$ *into a set of promises* $T_p^{i,1} \dots, T_p^{i,k}$ *containing exactly one joint transition for each state before checking that all pairs in* $\{(T_c^i, T_p^{i,1}), \dots, (T_c^i, T_p^{i,k})\}$ *are local progress properties*[5].

[4] For a transition labeled by p_γ with $\gamma \in \mathcal{I}_{obs}$, only the conditions on the local guard and function of the components involved in the interaction are kept. The guard and function of a τ-transition are the corresponding local guard and function.

[5] This is necessary to avoid that different processes choose a different joint transition in a given initial state.

Refinement. Refinement under context ensures that in the given context $(E, \mathcal{I})$ — and in any context refining it — safety and progress properties are preserved from the abstract component K_{abs} to the refined component K_{conc}. Moreover, refinement under context allows circular reasoning for the considered composition operators (provided that the assumptions are deterministic), because enabledness of transitions must be preserved from K_{conc} to K_{abs}. But only states reachable in the considered context must be related. To simplify the definition, we suppose that (a) K_{abs} has no internal transitions, (b) E has no transitions that it may do alone and (c) progress is refined without taking into account the context. The first two steps imply no loss of generality. The last simplification is sufficient for the considered application. It could be refined by requiring from K_{conc} only (part of) the progress properties of K_{conc} which are meaningful in $(E, \mathcal{I})$.

Refinement is defined by means of two relations (1) α relating variables of K_{conc} and K_{abs}, and (2) $\mathcal{R}$ relating concrete and abstract states. For preserving progress, we project transition sets of K_{abs} onto K_{conc} — for this purpose, we define the following auxiliary notations.

Definition 14 (Projection). *Let $\mathcal{R}$ be a relation on $(Q_{conc} \times Q_E) \times Q_{abs}$. We define the projection $\overline{\mathcal{R}}$ of $\mathcal{R}$ onto $Q_{conc} \times Q_{abs}$ by $q_c\,\overline{\mathcal{R}}\,q_a$ iff $\exists q_E$ s.t. $(q_c, q_E)\,\mathcal{R}\,q_a$. For $Q_a \subseteq Q_{abs}$, we denote $\overline{\mathcal{R}}^{-1}(Q_a) \subseteq Q_{conc}$ the inverse image of Q_a under $\overline{\mathcal{R}}$. $\overline{\mathcal{R}}^{-1}(\{q_a \overset{p}{\hookrightarrow}_{abs} q'_a\})$ denotes the set of p-transitions of K_{conc} between states in $\overline{\mathcal{R}}^{-1}(\{q_a\})$ and in $\overline{\mathcal{R}}^{-1}(\{q'_a\})$. This notation extends naturally to transition sets.*

Definition 15 (Refinement under context). *Given a relation α on $X_{conc} \cup X_{abs}$, K_{conc} refines K_{abs} in the context of $(E, \mathcal{I})$, denoted $K_{conc} \sqsubseteq_{E,\mathcal{I}} K_{abs}$, iff:*

(a) $\exists \mathcal{R} \subseteq (Q_{conc} \times Q_E) \times Q_{abs}$ s.t. $(q_c^0, q_E^0)\,\mathcal{R}\,q_a^0$ and s.t. $(q_c, q_E)\,\mathcal{R}\,q_a$ implies:

1. *$Inv_{q_c} \wedge \alpha(X_{conc}, X_{abs}) \implies Inv_{q_a}$*
2. *$\forall p[x] \in P$, the following holds ($\mathcal{V}_i$ denotes a valuation of X_i):*
   - *for any value v of x: $\exists t_c = q_c \xrightarrow{p}_c q'_c$ and $\mathcal{V}_c, \mathcal{V}_c^{new}$ satisfying Sem_{t_c} implies $\exists q'_a, t_a = q_a \xrightarrow{p}_a q'_a$ and $\mathcal{V}_a, \mathcal{V}_a^{new}$ consistent with α and satisfying Sem_{t_a}.*
   - *$\exists \gamma.\ P_\gamma = \{p, e\} \wedge (q_c, q_E) \overset{p_\gamma}{\hookrightarrow} (q'_c, q'_E) \implies (q'_c, q'_E)\,\mathcal{R}\,q'_a$ with q'_a as above[6].*
3. *$q_c \overset{\tau}{\hookrightarrow}_c q'_c \Longrightarrow (q'_c, q_E)\,\mathcal{R}\,q_a$: states related by τ-transitions refine the same state*

(b) The inverse image under $\overline{\mathcal{R}}$ of any progress property $pr = (T_c, T_p)$ of K_{abs}, which is $(\overline{\mathcal{R}}^{-1}(T_c), \overline{\mathcal{R}}^{-1}(T_p))$, is a progress property of K_{conc}.

Condition (a) ensures that refining an abstract component preserves safety properties. Condition (b) ensures preservation of progress properties. The last step to obtain a contract framework is to define conformance.

Definition 16 (Conformance). *Let $K_\perp = (TS, X, Inv, Prog)$ be defined as: $TS = (\{q_0\}, q_0, \emptyset, \emptyset)$, $X = \emptyset$, $I = \top$ and $Prog = \emptyset$. We define conformance as refinement in the context of $(K_\perp, \emptyset)$ — i.e., an "empty" with no connectors.*

Theorem 3. *We have defined a contract framework. Furthermore, if assumptions are deterministic, then circular reasoning is sound. See [10] for a proof.*

---

[6] If t is independent of the context, i.e., if $P_\gamma = \{p\}$, we use the convention $q_E \overset{\emptyset}{\hookrightarrow}_E q_E$.

4 An Application to Resource Sharing in a Network

We apply the proposed methodology to an algorithm for sharing resources in a network presented in [14]. The starting point is both a high-level property and an abstract description of the behavior of an individual node. We represent networks of arbitrary size by a grammar and associating a contract with each node, such that the correctness proof boils down to a set of small verification steps. We consider networks structured as binary trees defining a token ring[7].

Resources shared between nodes are represented by *tokens* circulating in packets containing one or more tokens along the token ring (see figure 3). The *value* of a packet is the number of tokens it contains. A particular token is the *privilege* — denoted P — which allows nodes to accumulate tokens.

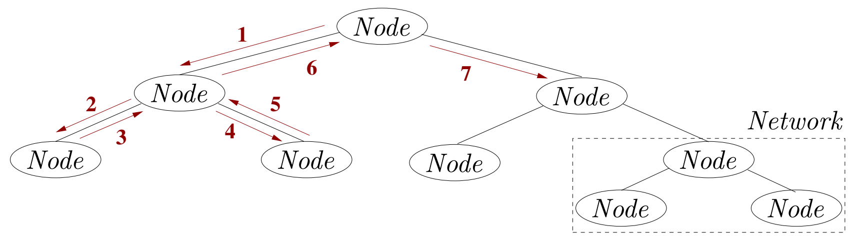


Fig. 3. The overall structure of the application

A node may *request* tokens (*Req* indicates the numbers of tokens requested). When it has enough tokens for satisfying its request, it is expected to *use* them, and relax the privilege if it has it; when it has resources (tokens) in use, it cannot request additional ones; it may later *free* them or keep them forever. A node can rise a request only when it has no resource in use and no pending request.

Tokens (and the privilege) circulate through ports called get_T (get_P) and $give_T$ ($give_P$), whereas the request, usage or freeing of tokens is indicated through observation ports req, use, $free$. Moreover, a *Node* has state variables indicating whether it has the privilege (P), its number of tokens (Tk), requests (Req) and some port variables used during interactions.

The network is defined by the grammar $\mathcal{G}$, where $\{E_\perp, Node\}$ are terminals and $\{Sys, Net\}$ nonterminals with axiom Sys. The rules are:

$$Sys \longrightarrow \mathcal{I}_{Net}\{E_\perp, Net), Net \longrightarrow Node, Net \longrightarrow \mathcal{I}\{Node, Net, Net\}$$

The connectors of the composition operators $\mathcal{I}$ and $\mathcal{I}_{Net}$ are indicated in Figures 4 and 5. They handle exchange of tokens and privileges and the observation of requested, used, respectively freed tokens.

We assume that connectivity of the network is guaranteed and tokens are never lost. Here, this assumption is encoded in the composition operator. This allows separating completely design and correctness proofs from the resource sharing algorithm and the algorithm guaranteeing connectivity, which is typically implemented in a lower layer of the overall network protocol.

[7] We restrict ourselves to binary branching for simplifying the presentation.

A top-level requirement φ. We consider here one of the top-level requirements of the algorithm, a progress requirement φ stating that "as long as the requests are reasonable, some of the nodes will be served" — *use* will occur — from time to time. φ is represented in our formalism as depicted in Figure 4, where the second progress property pr_2 says that "it is not possible to switch infinitely often between states S_1 and S_2 (that is, *free* occurs infinitely often) without that a *use* occurs infinitely often as well". "Reasonable" requests means that $0 < R^x \leq Tk$ where R^x is the maximal request and Tk the number of available tokens in the system.

Methodology. Our goal is to prove that every network built according to grammar $\mathcal{G}$, together with an environment $E_\perp$ giving back tokens and privilege immediately, conforms to φ. For this purpose, we instantiate the methodology of Section 2:

1. We define $\mathcal{C}_{\text{Node}} = (A_{\text{Node}}, \mathcal{I}_{\text{Node}}, G_{\text{Node}})$ and $\mathcal{C}_{\text{Net}} = (A_{\text{Net}}, \mathcal{I}_{\text{Net}}, G_{\text{Net}})$ that are contracts for component types *Net* and *Node*.
2. We show that $\mathcal{I}_{\text{Net}}\{A_{\text{Net}}, G_{\text{Net}}\} \preccurlyeq \varphi$.
3. We show that $\{\mathcal{C}_{\text{Node}},\ \mathcal{C}_{\text{Net}},\ \mathcal{C}_{\text{Net}}\}$ dominates $\mathcal{C}_{\text{Net}}$ w.r.t. $\mathcal{I}$.
4. We prove that $E_\perp$ satisfies $\mathcal{C}_{\text{Net}}^{-1}$ and that *Node* satisfies $\mathcal{C}_{\text{Node}}$ and $\mathcal{C}_{\text{Net}}$.

Note that if we want to further refine the *Node* component, we may start by a contract $\mathcal{C}_{\text{Node}} = (A_{\text{Net}},\ \mathcal{I}_{Net},\ Node)$. Now, let us give some details.

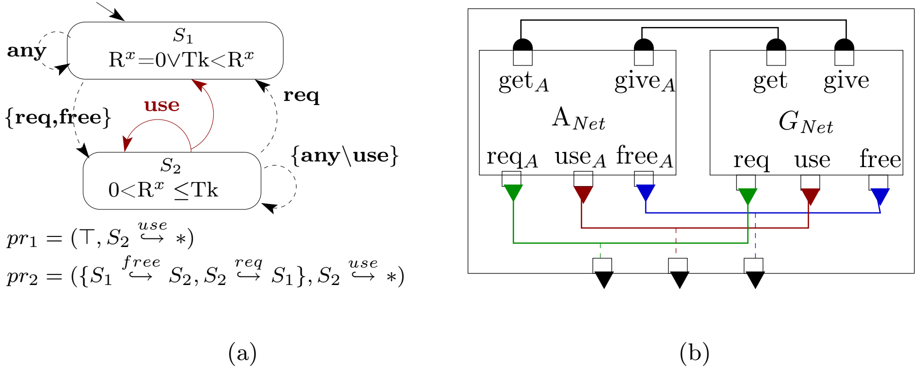


Fig. 4. (a) Top-level requirement φ; (b) Composition $\mathcal{I}_{\text{Net}}$ for contract $\mathcal{C}_{\text{Net}}$

Interaction models. Figure 4(b) shows the interaction model $\mathcal{I}_{\text{Net}}$ relating a network — and therefore also a leaf node — to the rest of the system. We represent by *get* and *give* respectively port sets $\{get_T, get_P\}$ and $\{give_T, give_P\}$ for token and privilege exchange. $\mathcal{I}$ consists of 3 observation connectors which export a *use* interaction of either the Net or its environment as a global *use* interaction, and analogously for the others. There are also 4 internal connectors for exchanging tokens and privilege. For example, connector

$\{give_T[tk] \mid get_{TA}[tk_A], \delta_G : [tk > 0], tk_A := tk\}$ pushes a positive number of tokens from the Network to the environment.

Due to lack of space, we do not present the assumptions and guarantees of the node and network contracts. They are detailed in [10].

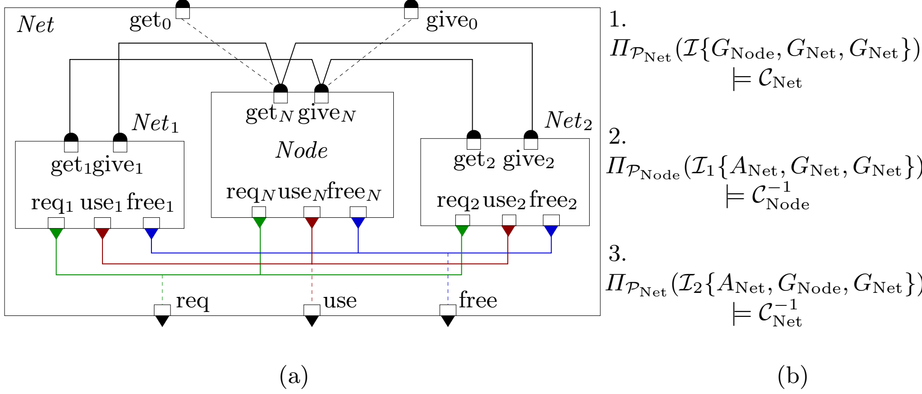


Fig. 5. (a) Structure of a network component; (b) Sufficient conditions for dominance

Figure 5(a) shows the inner structure of a network component Net. The interaction model $\mathcal{I}$ builds a tree from a (root) node[8] and two networks Net_1, Net_2. Interactions performed by the connectors depicted here are similar to those of Figure 4(b), except that they also ensure that tokens circulate in the correct order.

Experimental results. To show that $\{\mathcal{C}_{\text{Node}}, \mathcal{C}_{\text{Net}}, \mathcal{C}_{\text{Net}}\}$ dominates $\mathcal{C}_{\text{Net}}$ w.r.t. $\mathcal{I}$, it is sufficient, according to the sufficient condition of section 2, to prove the conditions given in Figure 5(b). Dominance, conformance and satisfaction problems are reduced to refinement under context checked and discharged automatically by a *Java* tool returning either yes or a trace leading to the violation of refinement.

5 Discussion and Future Work

We proposed a design and verification methodology which allows design and verification of safety and progress properties of distributed systems of arbitrary size. This methodology has been successfully applied to an algorithm for sharing resources in a tree-shaped network by automatically discharging the required conformance, dominance and satisfaction checks with a prototype tool.

There are several interesting directions to be explored: (a) We have excluded the use of contracts for assume/guarantee reasoning: we use contracts as design constraints for implementations which are maintained throughout the development and life cycle of the system. On the other hand, in assume/guarantee

[8] Which is connected in a slightly more complex manner than the leaf node.

based compositional verification, assumptions are used to deduce global properties (see [15]). We could integrate this into our methodology: as an example, in our network application, it would be enough to ensure that assumptions express sufficient progress to show conformance of a node contract to "node progress". (b) We would like to extend the methodology to multiple requirements, possibly by using a different decomposition of the system — i.e. a different grammar. (c) We also intend to extend the component framework to more general connectors and behaviors to express non functional properties. (d) We are currently considering building an efficient checker for different refinement relations, and then, implement tool support for the methodology. We also consider integration into a system design framework — such as SysML promoted by OMG.

References

1. Meyer, B.: Applying "design by contract". IEEE Computer 25(10), 40–51 (1992)
2. Quinton, S., Graf, S.: Contract-based verification of hierarchical systems of components. In: Proc. of SEFM 2008, pp. 377–381. IEEE Computer Society, Los Alamitos (2008)
3. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Specification, vol. 1. Springer, Heidelberg (1991)
4. Sifakis, J.: A framework for component-based construction. In: Proc. of SEFM 2005, pp. 293–300. IEEE Computer Society, Los Alamitos (2005)
5. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proc. of ESEC/SIGSOFT FSE 2001, pp. 109–120. ACM Press, New York (2001)
6. Bidinger, P., Stefani, J.B.: The Kell calculus: operational semantics and type systems. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 109–123. Springer, Heidelberg (2003)
7. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical Strucutres in Computer Science 14(3) (2004)
8. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Proc. of SEFM 2006, pp. 3–12. IEEE Computer Society, Los Alamitos (2006)
9. Benveniste, A., Caillaud, B., Ferrari, A., Mangeruca, L., Passerone, R., Sofronis, C.: Multiple viewpoint contract-based specification and design. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 200–225. Springer, Heidelberg (2008)
10. Ben-Hafaiedh, I., Graf, S., Quinton, S.: A contract framework for reasoning about safety and progress. Technical Report TR-2010-11, Verimag (2010)
11. Stadler, Z., Grumberg, O.: Network grammars, communication behaviours and automatic verification. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 68–80. Springer, Heidelberg (1990)
12. Bliudze, S., Sifakis, J.: The algebra of connectors: structuring interaction in BIP. In: Proc. of EMSOFT 2007, pp. 11–20. ACM Press, New York (2007)
13. Bozga, M., Jaber, M., Sifakis, J.: Source-to-source architecture transformation for performance optimization in BIP. In: Proc. of SIES 2009, pp. 152–160 (2009)
14. Datta, A.K., Devismes, S., Horn, F., Larmore, L.L.: Self-stabilizing k-out-of-l exclusion on tree network. CoRR abs/0812.1093 (2008)
15. de Roever, W.P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods, vol. 54. Cambridge University Press, Cambridge (2001)

Abstract Program Slicing: From Theory towards an Implementation

Isabella Mastroeni and Đurica Nikolić

Dipartimento di Informatica, Università di Verona, Italy
isabella.mastroeni@gmail.com, durica.nikolic@univr.it

Abstract. In this paper we extend the formal framework proposed by Binkley et al. for representing and comparing forms of program slicing. This framework describes many well-known forms of slicing in a unique formal structure based on (abstract) projections of state trajectories. We use this formal framework for defining a new technique of slicing, called abstract slicing, which aims to slice programs with respect to properties of variables. In this way we are able to extend the original work with three forms of abstract slicing, static, dynamic and conditioned, we show that all existing forms are instantiations of their corresponding abstract forms and we enrich the existing slicing technique hierarchy by inserting these abstract forms of slicing. Furthermore, we provide an algorithmic approach for extracting abstract slices. The algorithm is split into two modules: the simple approach, used for abstract static slicing, and the extended approach, composed of several applications of the simple one, which is used for abstract conditioned slicing.

Keywords: Program Slicing, Semantics, Program Analysis, Abstract Interpretation.

1 Introduction

It is well-known that as the size of programs increases, it becomes impractical to maintain them as monolithic structures. Indeed, splitting programs in smaller pieces allows to construct, understand and maintain large programs much more easily. *Program slicing* [3,7,14,15], is a program manipulation technique that extracts from programs statements which are relevant to a particular computation. In particular, a *program slice* is an executable program whose behavior must be identical to a specific subset of the original program's behavior. The specification of this behavior subset is called *slicing criterion* and can be expressed as the value of some sets of variables at some set of statements and/or program points [15]. Slicing[1] can be used in debugging [15], software maintenance [9], comprehension [4,8], re-engineering [5], etc.

Since the seminal paper introducing slicing [15], there have been many works proposing several notions of slicing, and different algorithms to compute slices (see [7,14] for good surveys about the existing slicing techniques). Program slicing is a transformation technique that reduces the size of programs to analyze.

[1] We use *slicing* (*slice*) and *program slicing* (*program slice*) as interchangeable terms.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 452–467, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Nevertheless, the reduction obtained by means of standard slicing techniques may be not sufficient for simplifying program analyses since it may keep more statements than those strictly necessary for the desired analysis. Suppose we are analyzing a program, and suppose we want a variable x to have a particular property φ. If we realize that, at a fixed program point, x does not have that property, we may want to understand which statements affect the computation of property φ of x, in order to find out more easily where the computation was wrong. In this case we are not interested in the exact value of x, hence we may not need *all* the statements that a standard slicing algorithm would return. In this situation we would need a technique that returns the minimal amount of statements that actually affect the computation of a desired property of x.

In this paper we introduce a novel notion of slicing, called *abstract slicing*, that looks for the statements affecting a fixed property (modelled in the context of abstract interpretation [6]) of variables of interest.

Example 1. Consider the program P in Fig. 1. If we are interested in exact values of variable d at the end of execution, program Q can be a *slice* of P with respect to that criterion. But, if we are interested in the parity of d at that point, the situation is a little bit different. The parity of d in $d = 2 * c + b + a - a$ depends on variable b only. Therefore, program R may be an *abstract slice* of P w.r.t. the specified criterion. Even in this simple case, the *abstract slice* gives us more precise information about the statements affecting the property of interest.

	Program P		Program Q		Program R
1	a := 1;	1	a := 1;	1	
2	b := b + 1;	2	b := b + 1;	2	b := b + 1;
3	c := c + 2;	3	c := c + 2;	3	
4	e := a + b + c;	4		4	
5	d := 2*c + b + a − a;	5	d := 2*c + b + a − a;	5	d := 2*c + b + a − a;

Fig. 1. Q and R are *slice* and *abstract slice* of P

Moreover, we have taken the first steps towards an implementation of this new form of slicing and we propose two approaches that, under certain hypotheses, extract *abstract slices*. We provide an example illustrating the application of the simplest approach and which highlights some of the differences between them.
This is not the first attempt to define weakened forms of slicing, often called abstract, even if it has never been formally described. Several authors focused on the concept of abstract dependency and tried to define it formally. Mastroeni and Zanardini [13] defined abstract slicing in terms of abstract dependency, a notion of dependency parametric on properties of interest, and obtained as negation of the notion of (abstract) non-interference [10]. The difference lies on the fact that while we provide a general formal definition, which allows several forms of abstract slicing simply by instantiating parameters, in [13] the authors consider only standard static slicing, defining abstract slicing by means of abstract dependencies. The notion of abstract slicing introduced by Hong, Lee and Sokolsky [11]

represents an implementation of a technique of conditioned slicing [4] based on abstract interpretation and model checking. Therefore, instead of weakening the observation of all executions, the authors only consider a subset of all possible executions.

2 Program Slicing

In this section we introduce different notions of program slicing and the Binkley et al. framework [1,2] in a slightly revised way by *constructing* a unified notation for slicing criteria.

Let us introduce some basic notions. We use $\mathbb{L}$ to denote the set of line numbers and $\mathbb{V}$ to denote a set of values. *Var* denotes a set of memory locations. A memory state is a function $\rho \in \mathbb{M}$, where $\mathbb{M} \stackrel{\text{def}}{=} Var \rightarrow \mathbb{V}$. When a program is in a state $\sigma_i = \langle n_i, k_i, \rho_i \rangle \in \mathbb{S} \stackrel{\text{def}}{=} \mathbb{L} \times \mathbb{N} \times \mathbb{M}$, it means that it is in a memory state ρ_i, the next statement to be executed is at a line number n_i and k_i is the number of occurrences of n_i until that point. A state trajectory $\sigma = (n_1, k_1, \rho_1) \dots (n_l, k_l, \rho_l) \in \Sigma \stackrel{\text{def}}{=} \mathbb{S}^*$ is a sequence of states $\sigma_i \in \mathbb{S}$, $i \in [1..l]$ a program goes through during the execution. The state trajectory obtained by executing program P from input state ρ is denoted T_P^ρ.

The very first definition of slicing was given by Weiser [15]. Nowadays, that technique is known as static slicing. It states that, if we execute both the original program and its static slice from the same input, any time the point of interest is reached in the original program, it is reached in the slice as well, and the values of all variables of interest in both programs are equal.

A key notion in program slicing is the *slicing criterion*, let us define a general characterization able to model all the forms of slicing we are going to introduce. As far as static slicing is concerned, the slicing criterion simply specifies the set $V \subseteq Var$ of variables of interest, and the program point $n \in \mathbb{L}$ of interest, namely the criterion is $\mathcal{C} = (V, n)$. Korel and Laski proposed a new techinque of slicing, called *dynamic slicing* [12]. It considers only one particular input, and the dynamic slice preserves the meaning of the original program for that input only. Let's consider two programs P and Q. I_P and I_Q denote sequences of line numbers reached during the executions of P and Q from ρ. The occurrence of interest is the n-th element of I_P. Q is a dynamic slice of P if there is an execution position n' in I_Q such that: 1) If we consider only first n elements of I_P, and if we eliminate all of them not appearing in Q, we obtain first n' elements of I_Q; 2) For all variables $v \in V$ the value of v in P before executing statement $I_P(n)$ is equal to the value of v in Q before executing statement $I_Q(n')$ and 3) Statements $I_P(n)$ and $I_Q(n')$ are equal. In order to characterize the slicing criterion also for dynamic slicing we have to enrich the notion and to consider a set of initial memories $\mathcal{I} \subseteq \mathbb{M}$. Hence the criterion is now $\mathcal{C} = (\mathcal{I}, V, n)$, where $\mathcal{I} = \mathbb{M}$ for static slicing, while $\mathcal{I} = \{\rho\}$, with $\rho \in \mathbb{M}$ for dynamic slicing.

Finally, Canfora et al. proposed a new techinque of slicing called *conditioned slicing* [4], which requires that a conditioned slice preserves the meaning of the original program for a set of inputs satisfying one particular condition π. Let us

denote $\mathbb{M}_{in}$ the set of input states satisfying π [1], then the slicing criterion is still $\mathcal{C} = (\mathcal{I}, V, n)$ where $\mathcal{I} = \mathbb{M}_{in}$ characterizes conditioned slicing.

Each type of slicing can have four forms: *standard* form - it considers one ore more points in a program with respect to a set of variables; *Korel and Laski* form (*KL*) - a stronger form where the program and the slice must follow identical paths; *iteration count* form (*IC*) - requires that a program and its slice need only agree at a particular iteration of a program point; and *Korel and Laski iteration count* form (*KLi*) - requires that a program and its slice must follow identical paths and need only agree at a particular iteration of a point. As far as the slicing criterion is concerned, we have to make some more changes in the definition. In particular, for the *KL* form of slicing we simply add a boolean parameter specifying whether we are considering a *KL* form or not, i.e., $\mathcal{C} = (\mathcal{I}, V, n, \mathcal{L})$, where $\mathcal{L} = true$ if we are considering a *KL* form of slicing, it is *false* otherwise. While, in order to model the *IC* form we have to change the third parameter of the criterion, in particular let $k \in \mathbb{N}$ be the iteration of the program point $n \in \mathbb{L}$ we are interested in, then instead of n in the criterion we should have $\langle n, k\rangle$. Hence, $\mathcal{C} = (\mathcal{I}, V, \mathcal{O}, \mathcal{L})$, where $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$. Note that $\{n\} \times \mathbb{N}$ represents the fact that we are interested in all occurrences of n.

There are also some *simultaneous* (sim.) types of slicing that consider more points of interest, and at each of them the slice has to preserve the values of all variables of interest. In order to deal also with sim. forms of slicing we have simply to extend the definition of slicing criterion by considering $\mathcal{O}$ a set instead of a singleton, namely $\mathcal{O} \in \wp(\mathbb{L} \times \wp(\mathbb{N}))$.

The formal framework proposed in [1,2] represents different forms of slicing by means of a pair: a *syntactic preorder*, a *function from slicing criteria to semantic equivalences*. The *preorder* fixes a syntactic relation between the program and its slices. In standard slicing, this relation represents the fact that slices are obtained from the original program by removing zero or more statements. This preorder is called *traditional syntactic ordering*, denoted $\sqsubseteq$, and is defined as follows: $Q \sqsubseteq P \Leftrightarrow \mathcal{F}(Q) \subseteq \mathcal{F}(P)$, where $\mathcal{F}(P)$ maps l to c if and only if program P contains the statement c at line number l.

The *function* fixes the semantic constraints that a subprogram has to respect in order to be a slice of the original program. It is worth noting that the equivalence relation returned by the function is uniquely determined by the form of slicing and by the chosen slicing criterion. In this way Binkley et al. are able to characterize eight forms of non-sim. slicing, and twelve forms of sim. slicing.

Finally, this framework is used to formally compare notions of slicing. Given two semantic equivalence relations $\approx_A$ and $\approx_B$, we say that $\approx_A$ subsumes $\approx_B$ if and only if for every two programs P and Q, $P \approx_B Q \Rightarrow P \approx_A Q$.

Given a binary relation on slicing criteria $\rightarrow$, we say that a form of slicing $(\sqsubseteq, \mathcal{E}_A)$ is *weaker than* $(\sqsubseteq, \mathcal{E}_B)$ w.r.t. $\rightarrow$ iff $\forall \mathcal{C}_A, \mathcal{C}_B$, slicing criteria such that $\mathcal{C}_A \rightarrow \mathcal{C}_B$, and $\forall P, Q$, if Q is a slice of P w.r.t. $(\sqsubseteq, \mathcal{E}_B(\mathcal{C}_B))$, then Q is a slice of P w.r.t. $(\sqsubseteq, \mathcal{E}_A(\mathcal{C}_A))$ as well, formally $(\sqsubseteq, \mathcal{E}_A) \overrightarrow{\subseteq} (\sqsubseteq, \mathcal{E}_B)$. Following this definition Binkley et al. show that all forms of slicing introduced in [1] are comparable in the way shown in Fig. 2. The symbols $\mathcal{S}$, $\mathcal{C}$, $\mathcal{D}$, $\mathcal{SS}$, $\mathcal{SC}$ and $\mathcal{SD}$ represent static,

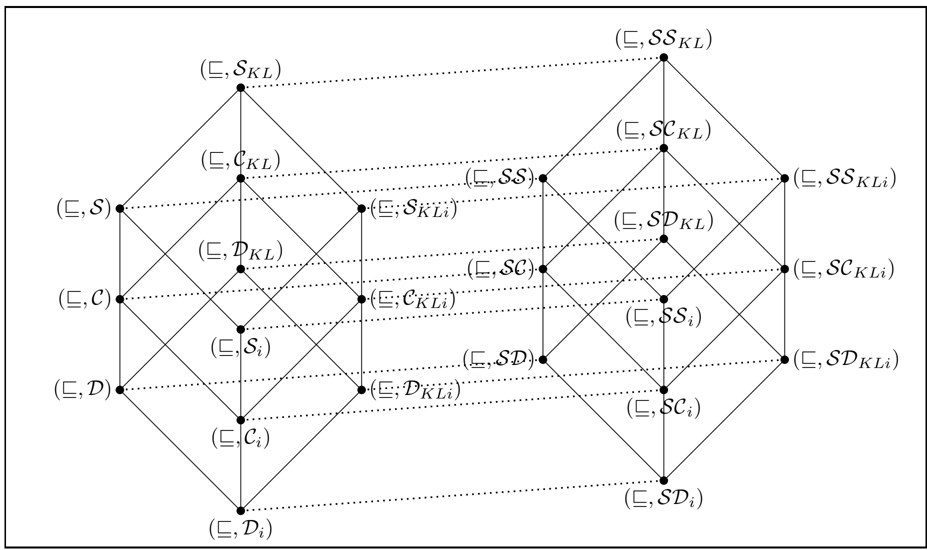


Fig. 2. Given two forms A and B, both (non-)sim., A is weaker than B if A is connected to B by a solid line and it is below B. If A is non-sim. and B is sim., then A is weaker than B if A is connected to B by a dotted line and it is to the left of B.

conditioned, dynamic, static sim., conditioned sim. and dynamic sim. types of slicing respectively. The subscripts i, KL and KLi represent IC, KL and KLi forms of slicing respectively, no subscript denotes standard forms of slicing. Following their structure, we enrich one of the slicing criterion comparison relations and consequently we insert the four non-sim. forms of conditioned slicing in the hierarchy constructed in [1], as shown in Fig. 2.

3 Abstract Program Slicing

Program slicing is used for reducing the size of programs to analyze. Nevertheless, sometimes this reduction is not sufficient for really improving the analyses. Suppose that some variables at some point of execution do not have a desired property. In order to understand where the error occurred it would be useful to find the statements affecting the property of these variables. Standard slicing may return too many statements, making it hard for the programmer to realize which ones caused the error. Consider the following example.

Example 2. Let us consider a program P in Fig. 3, that reverses a linked list. Lists are defined recursively with selectors *data*, storing the information, and *next*, pointing to the following element. Suppose a property of *well-formedness* is defined over lists. A list is well-formed if it has $data = [0]$ in the last element. A well-formed empty list is presented as $\langle [0] \rangle$, where square brackets indicate that 0 is not a proper element. Program P reverses a list l passed to it as argument, so if $l = \langle 1, 2, 3, 4, [0] \rangle$, the correct implementation of P should return $\langle 4, 3, 2, 1, [0] \rangle$.

```
1 list rev(list l) {
2   list *last;
3   list *tmp;
4   while (l->next != null){
5     tmp = l->next;
6     l->next = last;
7     last = l;
8     l = tmp;
9   }
10  return last;
11}
```

Fig. 3.

After running the program, we can realize that at line 9, list *last* is not well-defined. Standard static slicing w.r.t. criterion $(\{last\}, 9)$ would return the whole program as slice, since *last* is affected, even if not directly, by all statements. But if we use the property of *well-formedness*, and if we want to understand if *last* is well-formed, a slicing based on that abstract criterion should return the following slice: *list* $*$*last*; **return** *last*;, since the *well-formedness* property of *last* is not affected by the *while*. In this way we are able to characterize that the error occurred at line 2.

Let us introduce a new technique, called *abstract slicing*, which, regarding the syntax, can only delete statements from programs, while, regarding the semantics, compares the program and its abstract slices by considering *properties* instead of exact values of some variables. These properties are represented as abstract domains of the variable domain in the context of abstract interpretation [6]. The *abstract slicing* should help us finding all the statements affecting some particular *properties* of variables of interest.

First of all, we introduce the notion of abstract slicing criterion where we specify also the *property* of interest. For the sake of simplicity we define the abstract slicing criterion (Def. 1) only for non-sim. forms, i.e., $\mathcal{O}$ is a singleton and not a set of occurrences ($\mathcal{O} = \mathbb{L} \times \wp(\mathbb{N})$). In order to be as general as possible, we consider relational properties of variables and, for this reason, properties are associated with tuples and not with single variables.

Definition 1. *Let $\mathcal{I} \subseteq \mathbb{M}$ be a set of input memories, $V \subseteq Var$ be a set of variables of interest, $\mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N})$ be a set of occurrences of interest, $\mathcal{L}$ a boolean value determining KL forms. Let $V_1, \ldots, V_k$ be a partition of V and for each $i \in [1..k]$ let φ_i be a property of interest (modelled as an uco [6]) for V_i, then the abstract slicing criterion is $\mathcal{C}_\mathcal{A} = (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A})$, where $\mathcal{V} \stackrel{def}{=} \langle V_1, \ldots, V_k \rangle$ and $\mathcal{A} \stackrel{def}{=} \langle \varphi_1, \ldots, \varphi_k \rangle$.*

The extension of this definition to sim. forms is possible by considering $\mathcal{A}$ as a partial function that takes the set of points/occurrences of interest, the corresponding tuples of variables of interest $\mathcal{V}$, and return an abstract property to observe on $\mathcal{V}$ in the specified point/occurrence. For this reason in the following we consider any form of slicing. Note that, when dealing with non-abstract notions of slicing we have $\mathcal{A} = \langle \varphi_{\text{ID}}, \ldots, \varphi_{\text{ID}} \rangle \stackrel{\text{def}}{=} \mathcal{ID}$, where $\varphi_{\text{ID}} \stackrel{\text{def}}{=} \lambda x.x$.

It is worth underlying that, exactly as it happens for the non-abstract forms, we have that if $\mathcal{I} = \mathbb{M}$ we have *abstract static slicing*, if $|\mathcal{I}| = 1$ we have *abstract dynamic slicing*, otherwise we have *abstract conditioned slicing* .

Definition 2. *Let P and Q be executable programs such that Q can be obtained from P by removing zero or more statements and let $\mathcal{C}_\mathcal{A} = (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A})$. Q is an* abstract slice *of P w.r.t. $\mathcal{C}_\mathcal{A}$ if for each $\rho \in \mathcal{I}$, when the execution of P from input ρ reaches the point (or ocurrence) $\mathcal{O}$, the execution of Q from ρ reaches $\mathcal{O}$ as well, and for each $i \in [1..k]$, V_i has the same property φ_i both in P and in Q. Moreover, if $\mathcal{L} = true$ then P and Q have to follow identical paths [12].The extraction of* abstract slices *is called* abstract slicing*.*

3.1 Abstract Formal Framework

In this section we define the notion of abstract formal framework in which all forms of slicing can be formally represented. It is an extension of a mathematical structure introduced by Binkley et al. [1,2], that is used for representing and comparing different forms of slicing.

Following the framework of Binkley et al., described in Sect. 2, we represent an abstract form of slicing by a pair $(\sqsubseteq, \mathcal{E}_\mathcal{A})$, where $\sqsubseteq$ is the traditional syntactic ordering and $\mathcal{E}_\mathcal{A}$ is a function that maps abstract slicing criteria to semantic equivalence relations on programs. Given two programs P and Q, and an abstract slicing criterion $\mathcal{C}_\mathcal{A}$ we say that Q is $(\sqsubseteq, \mathcal{E}_\mathcal{A})$-*(abstract) slice* of P with respect to $\mathcal{C}_\mathcal{A}$ iff $Q \sqsubseteq P$ and $Q\ \mathcal{E}_\mathcal{A}(\mathcal{C}_\mathcal{A})\ P$. At this point we have to define the function $\mathcal{E}_\mathcal{A}$ in the context of *abstract slicing*. In order to derive this equivalence we have to define some preliminary useful notions.

Let us first define the *abstract memory state* which restricts the domain of a memory state to variables of interest only, and assigns to each tuple an abstract value determined by its abstract property of interest.

Definition 3. *Let $\rho \in \mathbb{M}$ be a memory state, $\mathcal{V} = \langle V_1, \ldots, V_k \rangle$ a tuple of variables and $\mathcal{A} = \langle \varphi_1, \ldots, \varphi_k \rangle$ the corresponding tuple of properties of interest. The* abstract memory state *is $\rho \restriction^\alpha \mathcal{V} \stackrel{def}{=} \mathcal{A} \circ \rho(\mathcal{V}) \stackrel{def}{=} \langle \varphi_1 \circ \rho(V_1), \ldots, \varphi_k \circ \rho(V_k) \rangle$.*

Consider the following example.

Example 3. Let $Var = \{x_1, x_2, x_3, x_4\}$ be a set of variables and suppose the properties of interest are the sign of $x_1 \times x_2$ and the parity of x_3. Let us consider *ucos* $\varphi_{\text{SIGN}} = \{(\varnothing, \varnothing), (\mathbb{Z}^+, \mathbb{Z}^-) \cup (\mathbb{Z}^-, \mathbb{Z}^+), (\mathbb{Z}^+, \mathbb{Z}^+) \cup (\mathbb{Z}^-, \mathbb{Z}^-), (\mathbb{Z}, \mathbb{Z})\}$ and $\varphi_{\text{PAR}} = \{\varnothing, 2\mathbb{Z}, 2\mathbb{Z}+1, \mathbb{Z}\}$. φ_{PAR} is defined on single integer variables v and represents their parity, i.e., if the value of v is even (odd), then it is mapped to all even (odd) numbers, $2\mathbb{Z}$ $(2\mathbb{Z}+1)$. φ_{SIGN} is defined on pairs of integer variables $\langle v, t \rangle$ and represents the sign of their product, i.e., if v and t are both positive or both negative, the pair is mapped to $(\mathbb{Z}^+, \mathbb{Z}^+) \cup (\mathbb{Z}^-, \mathbb{Z}^-)$ meaning that $v \times t$ is positive, otherwise it is mapped to $(\mathbb{Z}^+, \mathbb{Z}^-) \cup (\mathbb{Z}^-, \mathbb{Z}^+)$ meaning that $v \times t$ is negative.

Therefore, we consider $\mathcal{V} = \langle \{x_1, x_2\}, \{x_3\} \rangle$ and $\mathcal{A} = \langle \varphi_{\text{SIGN}}, \varphi_{\text{PAR}} \rangle$. Let $\rho = \langle 1, 2, 3, 4 \rangle$, then we obtain $\rho \restriction^\alpha \mathcal{V} = \mathcal{A} \circ \rho(\mathcal{V}) = \langle (\mathbb{Z}^+, \mathbb{Z}^+) \cup (\mathbb{Z}^-, \mathbb{Z}^-), 2\mathbb{Z}+1 \rangle$.

The *abstract projection* function modifies the state trajectory by removing all the states that do not contain occurrences or additional points of interest. If there is a state that contains an occurrence of interest, its memory state component is restricted to variables of interest, and for each tuple of interest only a property of interest is considered. We define an auxiliary function $Proj'^{\mathcal{A}}$ and the *abstract projection function*, $Proj^{\mathcal{A}}$ formally.

Definition 4. *Given $n \in \mathbb{L}$, $k \in \mathbb{N}$, $\rho \in \mathbb{M}$, and parameters $\mathcal{V}, \mathcal{O}, L, \mathcal{A}$, where $L \subseteq \mathbb{L}$ is a set of line numbers, we define a function $Proj'^{\mathcal{A}}$ as:*

$$Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n,k,\rho) \stackrel{def}{=} \begin{cases} (n, \rho \restriction^{\alpha} \mathcal{V}) & \text{if } (n,k) \in \mathcal{O}, \\ (n, \rho \restriction^{\alpha} \varnothing) & \text{if } (n,k) \notin \mathcal{O} \wedge n \in L \\ \lambda & \text{otherwise.} \end{cases}$$

$Proj'^{\alpha}$ takes a state of a state trajectory and returns a pair or an empty string, λ. If (n, k) is an occurrence of interest, i.e., if $(n, k) \in \mathcal{O}$, it returns a pair $(n, \rho \restriction^{\alpha} \mathcal{V})$. It means that at n we consider properties of interest for each tuple of interest, and not their exact values. If (n, k) is not an occurrence of interest, but n is an additional point of interest due to a *KL* form, i.e., if $n \in L$, $Proj'^{\alpha}$ returns a pair $(n, \rho \restriction^{\alpha} \varnothing)$. It means that we require the presence of n, but we are not interested in any property of any tuple, and therefore the domain of $\rho \restriction^{\alpha}$ is $\varnothing$.

Definition 5 (Abstract Projection). *Let $T = (n_1, k_1, \rho_1) \ldots (n_l, k_l, \rho_l) \stackrel{def}{=} \bigoplus_{i=1}^{l} (n_i, k_i, \rho_i)$ be a state trajectory, we define the notion of abstract projection as $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(T) \stackrel{def}{=} \bigoplus_{i=1}^{l} Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},L,\mathcal{A})}(n_i, k_i, \rho_i)$.*

The *abstract projection* function permits us to define all the semantic equivalence relations we need for representing the abstract forms of slicing: *abstract static backward equivalence*, *abstract dynamic backward equivalence* and *abstract conditioned backward equivalence*. If we have two programs P and Q, we can say that Q is an *abstract (static, dynamic or conditioned) slice* of P if $Q \sqsubseteq P$ and if Q is *abstract backward equivalent* to P w.r.t. the corresponding semantic equivalence relation.

3.2 Abstract Unified Equivalence

The only missing thing for completing the formal definitions of the three forms of abstract slicing is the characterization of the functions mapping slicing criteria to semantic equivalence relations. The *abstract unified equivalence*, $\mathcal{U}^{\mathcal{A}}$, is a function that takes parameters $\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}$ characterizing the slicing criterion and, therefore, the form of slicing, and returns a corresponding abstract semantic equivalence relation, which has to hold between a program and each one of its (abstract) slices. It can be used as a unified model for representing abstract equivalence relations for all possible forms of slicing.

Definition 6 ($\mathcal{U}^{\mathcal{A}}$). *Let P and Q be executable programs, $\mathcal{C}_{\mathcal{A}} = (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A})$ be an abstract criterion, and let L be such that $\mathsf{L}_{\mathcal{L}}(P, Q) = I_P \cap I_Q$, where $I_P \subseteq \wp(\mathbb{L})$*

denotes the set of all line numbers of P*, if* $\mathcal{L} = true$*,* $\mathsf{L}_{\mathcal{L}}(P,Q) = \varnothing$ *otherwise*[2]. *Then* P *is* abstract backward equivalent *to* Q*, denoted* $P\ \mathcal{U}^{\mathcal{A}}(\mathcal{I},\mathcal{V},\mathcal{O},\mathsf{L}_{\mathcal{L}},\mathcal{A})\ Q$*, iff* $\forall \rho \in \mathcal{I}. Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\mathsf{L}_{\mathcal{L}}(P,Q),\mathcal{A})}(T^{\rho}_{P}) = Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\mathsf{L}_{\mathcal{L}}(P,Q),\mathcal{A})}(T^{\rho}_{Q})$.

Now we are able to define the functions mapping each criterion $\mathcal{C}_{\mathcal{A}}$ to the corresponding semantic equivalence relation.

$$\mathcal{E}_{\mathcal{A}} \stackrel{\text{def}}{=} \lambda(\mathcal{I},\mathcal{V},\mathcal{O},\mathcal{L},\mathcal{A}).\, \mathcal{U}^{\mathcal{A}}(\mathcal{I},\mathcal{V},\mathcal{O},\mathsf{L}_{\mathcal{L}},\mathcal{A})$$

Hence a generic form of slicing can be represented as $(\sqsubseteq, \mathcal{E}_{\mathcal{A}})$. These functions can be used for the formal definitions of both abstract and non-abstract forms of slicing in the abstract formal framework. We can therefore state that the abstract formal framework is a generalization of the original formal framework. The following examples show how it is possible to use these definitions in order to check whether a program is an *abstract slice* of another one.

Program P:
```
1 read(n);
2 read(s);
3 i := 1;
4 while (i<=n) do
5    s := s + 2*i;
6    i := i+1;
7 od
```

Program Q:
```
1 read(n);
2 read(s);
3
4
5
6
7 end
```

Fig. 4.

Program R:
```
1 read(n);
2 s:=0;
3 i := 1;
4 while (i<=n) do
5    s := s + i;
6    i := i+1;
7 od
```

Program S:
```
1 read(n);
2 s:=0;
3
4
5
6
7 end
```

Fig. 5.

Example 4. Let us consider programs P and Q given in Fig. 4, and let $\mathcal{C}_{\mathcal{A}} = (\mathbb{M}, \langle s\rangle, \{7\}\times\mathbb{N}, \mathit{false}, \langle\varphi_{\text{PAR}}\rangle)$, i.e., we are interested in the parity ($\mathcal{A} = \langle\varphi_{\text{PAR}}\rangle$) of s ($\mathcal{V} = \langle s\rangle$) at the end of execution ($\mathcal{O} = \{7\}\times\mathbb{N}$) for all possible inputs ($\mathcal{I} = \mathbb{M}$). Since $Q \sqsubseteq P$, in order to show that Q is an *abstract static slice* of P w.r.t. $C_{\mathcal{A}}$, we have to show that $P\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ Q$ holds. Let $\rho = \{n \leftarrow a, s \leftarrow b\} \in \mathcal{I}$ be an initial memory. Execution of P from ρ determines:

$$\begin{array}{l} T^{\rho}_{P} = \ldots(5^1, \langle a, b, 1\rangle)(6^1, \langle a, b+2, 1\rangle)(4^2, \langle a, b+2, 2\rangle)\ldots \\ (5^a, \langle a, b+2+\ldots+2(a-1), a\rangle)(6^a, \langle a, b+2+\ldots+2a, a+1\rangle) \\ (4^{a+1}, \langle a, b+2+\ldots+2a, a+1\rangle)(7^1, \langle a, b+a(a+1), a+1\rangle), \end{array}$$

where a^b is the b-th occurrence of point a. Application of $Proj^{\alpha}$ to T^{ρ}_{P} gives:

$$\begin{array}{c} Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{P}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, b+a(a+1), a+1\rangle) \\ = (7, \langle a, b+a(a+1), a+1\rangle \upharpoonright^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(\langle b+a(a+1)\rangle)) \\ = (7, \varphi_{\text{PAR}}(b)). \end{array}$$

Let us give some clarifications: the only state of interest is the state containing the occurrence of the point 7, $(7^1, \langle a, b+a(a+1), a+1\rangle)$, so we apply the function

[2] When dealing with KL forms, i.e., $\mathcal{L} = true$ it takes the intersection of the line numbers, otherwise it is the empty set.

$Proj'^{\alpha}$ to that state. Since we have $7^1 = (7,1) \in \mathcal{O}$, $Proj'^{\alpha}$ returns $(7, \rho \restriction^{\alpha} \mathcal{V}) = (7, \langle a, b{+}a(a{+}1), a{+}1\rangle \restriction^{\alpha} \{\langle s\rangle\})$. The *abstract memory state* restricts the domain of ρ to variables of interest only, so we consider only the part of ρ regarding s, i.e., $b{+}a(a{+}1)$. Hence, the result of the last application is $(7, \mathcal{A}\circ\rho(\{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(\langle b{+}a(a{+}1)\rangle)))$. Since the parity of $b{+}a(a{+}1)$ depends on the parity of b only, the final result of the application of $Proj^{\alpha}$ is $(7, \varphi_{\text{PAR}}(b))$. Now, the execution of Q from ρ gives the following state trajectory: $T^{\rho}_{Q} = (1^1, \langle a\rangle)(2^1, \langle a, b\rangle)(7^1, \langle a, b\rangle)$. Application of $Proj^{\alpha}$ to T^{ρ}_{Q} gives: $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{Q}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, b\rangle) = (7, \langle a, b\rangle \restriction^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(b))$, and therefore $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{P}) = Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{Q})$. As ρ is an arbitrary input, we realize that this equation holds for each $\rho \in \mathcal{I}$, hence $P\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ Q$, and this implies that Q is an *abstract static slice* of P w.r.t. $\mathcal{C}_{\mathcal{A}}$.

Example 5. Consider R and S in Fig. 5, and let $\mathcal{C}_{\mathcal{A}} = (\mathcal{I}, \langle s\rangle, \{7\}\times\mathbb{N}, \mathit{false}, \langle\varphi_{\text{PAR}}\rangle)$, where $\mathcal{I} = \{\rho \mid \rho(n) \in 4\mathbb{Z}\}$, i.e., we are interested in the parity of s at the end of execution for all inputs that assign to n a multiple of 4. Since $S \sqsubseteq R$, in order to show that S is an *abstract conditioned slice* of R w.r.t. $\mathcal{C}_{\mathcal{A}}$, we have to show that $R\ \mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})\ S$ holds. Let $\rho \in \mathcal{I}$ be an initial memory, and suppose $\rho(n) = a = 4b$. Execution of R from ρ determines:

$$\begin{array}{c} T^{\rho}_{R} = \ldots(5^1, \langle a, 0, 1\rangle)(6^1, \langle a, 1, 1\rangle)(4^2, \langle a, 1, 2\rangle)\ldots \\ (5^a, \langle a, 1+2+\ldots+(a-1), a\rangle)(6^a, \langle a, 1+2+\ldots+a, a+1\rangle) \\ (4^{a+1}, \langle a, \frac{1}{2}a(a+1), a+1\rangle)(7^1, \langle a, \frac{1}{2}a(a+1), a+1\rangle), \end{array}$$

Application of $Proj^{\alpha}$ to T^{ρ}_{R} gives:

$$\begin{array}{c} Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{R}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, \frac{1}{2}a(a+1), a+1\rangle) \\ = (7, \langle a, \frac{1}{2}a(a+1), a+1\rangle \restriction^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(\langle \frac{1}{2}a(a+1)\rangle)) \\ = (7, \varphi_{\text{PAR}}(2b(4b+1))) = (7, 2\mathbb{Z}). \end{array}$$

Execution of S from ρ gives the state trajectory $T^{\rho}_{S} = (1^1, \langle a\rangle)(2^1, \langle a, 0\rangle)(7^1, \langle a, 0\rangle)$. Application of $Proj^{\alpha}$ to T^{ρ}_{S} gives: $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{S}) = Proj'^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(7^1, \langle a, 0\rangle) = (7, \langle a, 0\rangle \restriction^{\alpha} \{\langle s\rangle\}) = (7, \varphi_{\text{PAR}}(0))) = (7, 2\mathbb{Z})$, and therefore we have $Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{R}) = Proj^{\alpha}_{(\mathcal{V},\mathcal{O},\varnothing,\mathcal{A})}(T^{\rho}_{S})$. As ρ is an arbitrary input from $\mathcal{I}$, we realize that this equation holds for each $\rho \in \mathcal{I}$, hence $R\mathcal{E}_{\mathcal{A}}(\mathcal{C}_{\mathcal{A}})S$, and this implies that S is an *abstract conditioned slice* of R w.r.t. $\mathcal{C}_{\mathcal{A}}$.

4 Comparing Forms of Abstract Slicing

In this section we provide a formal theory which allows to compare the abstract forms of slicing among themselves. Moreover, in the same theory we can compare the abstract forms of slicing with the non-abstract ones. First of all, we show under which conditions an abstract semantic equivalence relation subsumes another one, and analogously we show when the form of (*abstract*) slicing corresponding to the former equivalence relation subsumes the form of (*abstract*) slicing corresponding to the latter one. These results are necessary for obtaining a precise characterization of the *weaker then* relation (Sect. 2) involving also the new forms of slicing.

The *abstract unified equivalence lemma* ($\mathcal{U}^{\mathcal{A}}$-lemma), shows under which conditions on the parameters determined by the slicing criteria, two semantic equivalence relations $\approx_A$ and $\approx_B$ are such that $\approx_A$ subsumes $\approx_B$.

Lemma 1 ($\mathcal{U}^{\mathcal{A}}$-lemma). *Given two forms of (abstract) slicing and their corresponding criteria* $\mathcal{C}^i_{\mathcal{A}} = (\mathcal{I}^i, \mathcal{V}^i, \mathcal{O}^i, \mathcal{L}^i, \mathcal{A}^i)$ *, with* $i \in \{1,2\}$*.*

If $\mathcal{I}^1 \subseteq \mathcal{I}^2, \mathcal{O}^1 \subseteq \mathcal{O}^2, \mathcal{V}^1 \subseteq \mathcal{V}^2, \forall P, Q. \mathsf{L}_{\mathcal{L}^1}(P,Q) \subseteq \mathsf{L}_{\mathcal{L}^2}(P,Q)$ *and* $\mathcal{A}^2 \overline{\sqsubseteq} \mathcal{A}^1$*, where* $\overline{\sqsubseteq}$ *denotes the approximation relation between tuples of abstractions*[3]*, then we have that* $P\ \mathcal{U}^{\mathcal{A}}(\mathcal{I}^2, \mathcal{V}^2, \mathcal{O}^2, \mathsf{L}_{\mathcal{L}^2}, \mathcal{A}^2)\ Q \Rightarrow P\ \mathcal{U}^{\mathcal{A}}(\mathcal{I}^1, \mathcal{V}^1, \mathcal{O}^1, \mathsf{L}_{\mathcal{L}^1}, \mathcal{A}^1)\ Q$*.*

This lemma tells us how it is possible to find the relationship (in the sense of subsume) between two semantic equivalence relations determined by two forms of (*abstract*) slicing. In particular, if we denote abstract notions of slicing by putting an $\mathcal{A}$, e.g., $\mathcal{AS}$ denotes static abstract slicing and $\mathcal{AD}$ denotes dynamic abstract slicing, by using this lemma we can show that, given a slicing criterion $\mathcal{C}_{\mathcal{A}}$, all the abstract equivalence relations introduced in Sect. 3.2, subsume the corresponding non-abstract equivalence relations $\mathcal{S}(\mathcal{C}_{\mathcal{A}})$, $\mathcal{D}(\mathcal{C}_{\mathcal{A}})$ and $\mathcal{C}(\mathcal{C}_{\mathcal{A}})$. Furthermore, by using this lemma we can show that $\mathcal{AD}(\mathcal{C}_{\mathcal{A}})$ subsumes $\mathcal{AC}(\mathcal{C}_{\mathcal{A}})$ which subsumes $\mathcal{AS}(\mathcal{C}_{\mathcal{A}})$. Hence, let us first recall an important result that we then apply in the context of abstract slicing, showing the relationship between the different forms of slicing.

Theorem 1. *[2] Let* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *be semantic equivalence relations such that* $\mathcal{R}_2$ *subsumes* $\mathcal{R}_1$*, then* $\forall P, Q$ *we have* $P\ (\sqsubseteq, \mathcal{R}_1)\ Q \ \Rightarrow\ P\ (\sqsubseteq, \mathcal{R}_2)\ Q$*.*

Hence, for instance, given two slicing criteria $\mathcal{C}'_{\mathcal{A}}$ and $\mathcal{C}''_{\mathcal{A}}$ satisfying hypotheses of Lemma 1, for each pair of programs P and Q:

$$P\ (\sqsubseteq, \mathcal{AS}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AC}(\mathcal{C}'_{\mathcal{A}}))\ Q \qquad P\ (\sqsubseteq, \mathcal{AC}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AD}(\mathcal{C}'_{\mathcal{A}}))\ Q$$
$$P\ (\sqsubseteq, \mathcal{S}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AS}(\mathcal{C}'_{\mathcal{A}}))\ Q \qquad P\ (\sqsubseteq, \mathcal{D}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AD}(\mathcal{C}'_{\mathcal{A}}))\ Q$$
$$P\ (\sqsubseteq, \mathcal{C}(\mathcal{C}''_{\mathcal{A}}))\ Q \Rightarrow P\ (\sqsubseteq, \mathcal{AC}(\mathcal{C}'_{\mathcal{A}}))\ Q$$

Finally, in order to formally prove the *weaker than* relation among the considered forms of slicing we have to define the slicing criteria comparison relation as introduced in Sect. 2. We define two slicing criterion comparison relations $\stackrel{\alpha_1}{\rightarrow}$ and $\stackrel{\alpha_2}{\rightarrow}$. $\stackrel{\alpha_1}{\rightarrow}$ permits to compare criteria of abstract forms of slicing among themselves, while $\stackrel{\alpha_2}{\rightarrow}$ permits to compare criteria of abstract forms of slicing with criteria of non-abstract forms of slicing.

Definition 7. *The slicing criterion comparison relation* $\stackrel{\alpha_1}{\rightarrow}$ *is defined by the following rule:* $\forall \mathcal{I}^1, \mathcal{I}^2 \subseteq \mathbb{M}$ *such that* $\mathcal{I}^1 \subseteq \mathcal{I}^2$*,* $\forall \mathcal{V} = \langle V_1, \ldots, V_k \rangle, \forall \mathcal{A} = \langle \varphi_1, \ldots, \varphi_k \rangle, \forall \mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N}), \mathcal{L} \in \{true, false\}$

$$(\mathcal{I}^1, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}) \stackrel{\alpha_1}{\rightarrow} (\mathcal{I}^2, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}).$$

For instance, this rule permits us to compare $(\sqsubseteq, \mathcal{AC})$ to $(\sqsubseteq, \mathcal{AS})$ being $\mathcal{I} \subseteq \mathbb{M}$, and $(\sqsubseteq, \mathcal{AD})$ to $(\sqsubseteq, \mathcal{AC})$ whenever $\rho \in \mathcal{I}$.

[3] We denote $\overline{\sqsubseteq}$ the relation "more concrete than" in the lattice of abstract interpretations between tuples of abstractions [6], i.e., $\mathcal{A}^1 \overline{\sqsubseteq} \mathcal{A}^2$ iff $\forall i \leq k.\ \varphi^1_i \sqsubseteq \varphi^2_i$ and $\varphi' \sqsubseteq \varphi''$ iff $\forall x.\ \varphi'(x) \leq \varphi''(x)$.

Definition 8. *The slicing criteria comparison relation* $\stackrel{\alpha_2}{\rightarrow}$ *is defined by the following rule:* $\forall \mathcal{I} \subseteq \mathbb{M}$, $\forall \mathcal{V} = \langle V_1, \ldots, V_k \rangle, \forall \mathcal{A} = \langle \varphi_1, \ldots, \varphi_k \rangle$ *where* $V = \bigcup_{i \in [1,k]} V_i$, $\forall \mathcal{O} \in \mathbb{L} \times \wp(\mathbb{N}), \mathcal{L} \in \{true, false\}$,

$$(\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{A}) \stackrel{\alpha_2}{\rightarrow} (\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}, \mathcal{ID}) = (\mathcal{I}, V, \mathcal{O}, \mathcal{L}).$$

This rules allows, for instance, to compare $(\sqsubseteq, \mathcal{AS})$ to $(\sqsubseteq, \mathcal{S})$, $(\sqsubseteq, \mathcal{AC})$ to $(\sqsubseteq, \mathcal{C})$ and $(\sqsubseteq, \mathcal{AD})$ to $(\sqsubseteq, \mathcal{D})$.

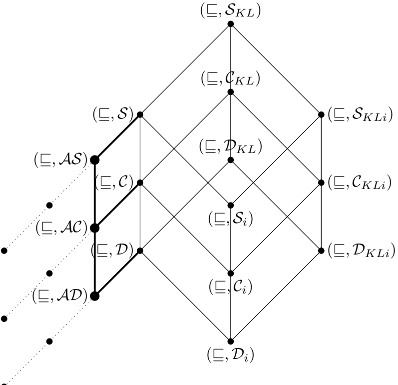


Fig. 6.

In Fig. 6 we show the non-sim. hierarchy obtained by enriching the hierarchy in Fig. 2 with standard forms of *abstract static slicing*, *abstract dynamic slicing* and *abstract conditioned slicing*. In general we can enrich this hierarchy with any abstract form of slicing simply by using the comparison notions defined above. Non-abstract forms are particular cases of abstract forms of slicing, as they can be instantiated by choosing the identity property, φ_{ID}, for each variable of interest. Hence, non-abstract forms are the "strongest" forms, since for each property φ, we have $\varphi_{\text{ID}} \sqsubseteq \varphi$. Moreover, if we fix the parameters $\mathcal{I}, \mathcal{V}, \mathcal{O}, \mathcal{L}$ of an abstract form, by abstracting the parameter $\mathcal{A}$, i.e., by reducing the information represented by the property, the abstract slicing forms we obtain become weaker. The dotted lines in Fig. 6 represent this fact.

5 Towards an Implementation

In this section, we describe an algorithmic approach for obtaining *abstract slices*. The idea is to define a notion of abstract state which observes variables of interest by means of abstract properties. These states are used for analyzing the *evolution* of the *properties* of variables of interest instead of their values. In order to perform the *evolution* analysis, we construct the *abstract* state graph (*ASG*), whose vertices are abstract states and which models program executions at some level of abstraction. At this point, we propose a technique for *pruning* the *ASG* in order to remove *all* the statements not relevant for the properties of interest.

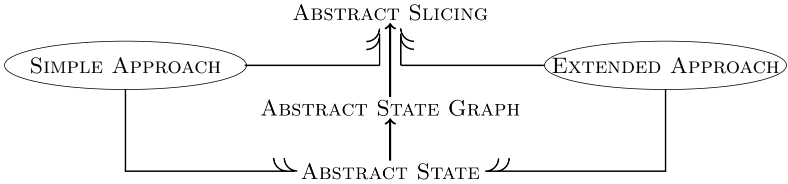


Fig. 7.

In the following we consider two different approaches: *simple* (*SA*) and *extended* (*EA*). The difference between them lies in the *abstract state* definition, where *EA* adds information about the relationship between input variables and properties of variables of interest. Moreover, the two approaches differ also in the pruning process. In particular, *EA*'s pruning consists of more than one application of *SA*'s pruning. See Fig. 7 for a graphical representation of our approaches. Note that *SA* can be used for extraction of *abstract static slices* only. Unfortunately, since it does not consider the relationship between variables of interest and input variables, it is not precise enough and cannot be applied to the problem of *abstract conditioned slicing*.

The *simple approach pruning* is illustrated by the method **pruningSA** given in Fig. 8, and we illustrate its application by the following example. Instead, the *extended approach pruning* is characterized by the method **pruningEA**, given in Fig. 10, and it will be only informally and intuitively described.

pruningSA(G)	**removeBlock**$(G,\ B)$	**findBlocks**(G)

Input: $G=(V,E)$ - ASG_S
Output: pruned G
$list$=**findBlocks**(G);
foreach $B \in list$
  if $(absv(in(B))=absv(out(B)))$
  then
    G=**removeBlock**$(G,\ B)$;
    $E=E\cup\{(in(B),out(B))\}$;
    $G=(V,E)$;
  fi
endf
return G;
	Input: $G=(V,E)$ - ASG_S,
	      $B\subseteq V$ - block
	Output: G' - ASG_S
	foreach $a\in B$
	  foreach $(a,b)\in E$
	    $E=E\setminus\{(a,b)\}$;
	  endf
	  foreach $(b,a)\in E$
	    $E=E\setminus\{(b,a)\}$;
	  endf
	  $V=V\setminus\{a\}$;
	endf
	return (V,E);
		Input: $G=(V,E)$ - ASG_S
		Output: list of blocks to be removed
		$list=\emptyset$;
		$G^{SCC}=(V^{SCC},E^{SCC})$=**tarjan**$(G)$;
		foreach component $V_i\in V^{SCC}$
		  $i=0$; $o=0$;
		  foreach $(V_i,V_j)\in E^{SCC}$ $o=o+1$; **endf**
		  foreach $(V_j,V_i)\in E^{SCC}$ $i=i+1$; **endf**
		  if $i=1\wedge o=1$ **then**
		    $list=list\cup\{C_i\}$
		  fi
		endf
		return $list$;

Fig. 8.

Example 6. Consider P in Fig. 4, and let $\mathcal{C}_\mathcal{A}=(\mathbb{M},\langle s\rangle,\{7\}\times\mathbb{N},\mathit{false},\langle\varphi_{\mathrm{PAR}}\rangle)$. In the *abstract states* induced by α, s can have two abstract values: `E`, if its concrete value is even, or `O`, if its concrete value is odd. In Fig. 9 we give the *ASG* of P for $\mathcal{C}_\mathcal{A}$, which vertices are identified by an overlined number given in the top left corner. Consider only edges represented by a solid line. Let us consider a block B composed of vertices $\overline{7}$, $\overline{9}$ and $\overline{11}$. The only input and output edges of B are edges from $\overline{5}$ and towards $\overline{13}$ respectively. The vertices $\overline{5}$ and $\overline{13}$ assign the same abstract value, `E`, to s. It means that we can remove the block B and add an

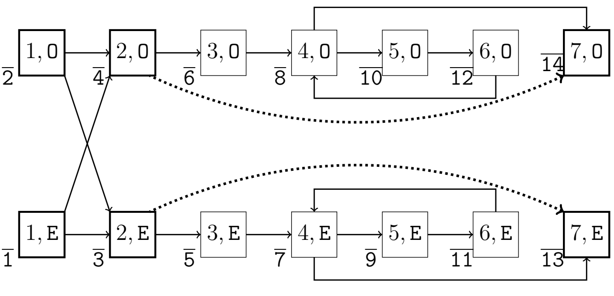


Fig. 9.

---

pruningEA(G)

Input: $G=[G_1,\ldots,G_w]$ - ASG_E
Output: pruned G
foreach $i \in [1..w]$
  $G'_i =$ **pruningSA**(G_i);
endf
return $G=[G'_1,\ldots,G'_w]$;

Fig. 10.

edge from $\overline{5}$ to $\overline{13}$. Analogously, it is possible to remove a block C, composed of vertices $\overline{8}$, $\overline{10}$ and $\overline{12}$, and to add an edge from $\overline{6}$ to $\overline{14}$. In a similar way we can remove vertices $\overline{5}$ and $\overline{6}$ and add the edges from $\overline{3}$ to $\overline{13}$ and from $\overline{4}$ to $\overline{14}$. These edges are represented by dotted lines in Fig. 9. The remaining vertices, $\overline{1}$, $\overline{2}$, $\overline{3}$ and $\overline{4}$[4], represented in bold in Fig 9, correspond to the statements 1 and 2, which form the *abstract slice*, Q, given in Fig. 4.

Abstract slices obtained this way can be larger than they are expected to be. In the worst case, we do not remove any instruction from the starting program P.

In order to extract *abstract conditioned slices*[5] we refine *abstract states*. This refinement permits us to construct ASG_E containing more information and which captures the relationship between input and properties of variables of interest. At this point the *extended approach pruning* (Fig. 10), takes as input G, and applies the method `pruningSA` (Fig. 8) to all subgraphs G_i, $i \in [1..w]$ of G. Each of these applications returns a pruned subgraph, G'_i, which permits us to construct an *abstract slice*, called *partial abstract slice*, containing only statements of P which vertices appear in G'_i. From the complexity point of view we can note that, once we have the graph, the complexity of both the algorithms is linear w.r.t. the dimension of the graph. Indeed, the most difficult part of our approaches is the construction of ASG. Note that the ASGs used in our examples are constructed manually. In order to completely automate our approaches, we should use some automatic tool for the constructions of our ASGs.

---

[4] Vertices $\overline{13}$ and $\overline{14}$ represent the end of execution.

[5] Recall that abstract dynamic slicing is a particular case of abstract conditioned slicing.

Yorsh et al. [16] presented a method for static program analysis that leverages tests and concrete program executions. They introduce *state abstractions* which generalize the set of program states obtained from concrete executions and define a notion of *abstract graphs*, similar to ours. Furthermore, they use a theorem prover to check that the generalized set of concrete states covers all potential executions and satisfies additional safety properties, and they use these results to construct an approximation of their *abstract graphs*. The relation between [16] and our work should be further analyzed, in order to use their method for an automatic construction of our graphs (ASG).

6 Conclusion and Future Work

This paper introduces a notion of *abstract slicing*. Abstract slicing is very useful when we want to determine which statements of original program affect some particular properties of variables of interest. We have defined the *Abstract Formal Framework*, that is an extension of a structure used for representation of all existing forms of slicing and for their comparison [1,2]. Within our framework it is possible to represent even the abstract forms of slicing. Furthermore, we formally proved that the three abstract forms are *weaker* than the corresponding standard forms.

Moreover, we have made the first steps towards implementation, and we introduced two novel approaches (*simple* and *extended*) for the extraction of *abstract slices*. We illustrated their application with an example showing the first approach, and informally explaining how this approach is used for the extended one. The automation of these approaches deserves further research. The main challenge is an automatic construction of ASG, and it might be done by using a theorem prover controlling whether an edge between two vertices exists or not. Unfortunately, this construction may introduce some edges that do not correspond to real cases, so the *abstract slices* may not be precise enough. We will try to solve the problem of automatic construction of ASG by using some sophisticated method, such as [16].

We think that the abstract slicing can be seen as a particular technique of *code obfuscation*. Suppose we have a program P and we want the obfuscated program $\mathcal{O}(P)$ to preserve properties of interest for some particular variables of P, but to hide everything else. For instance, if $x = 10$ and $y = 0$ in P, $\mathcal{O}(P)$ could tell us that $x \leq 20$ but it should not public neither its real value nor any information regarding y. In this case abstract slices of the original program w.r.t. a specified criterion can be very useful. Abstract slicing would be even more applicable to code obfuscation if we defined a notion of *abstract amorphous slicing*, that would permit us even to modify statements, and not to eliminate them only. The relationship between abstract slicing (standard or amorphous) and code obfuscation deserves further research. Moreover, we think there is a strong connection between *non-interference* and program slicing, and therefore between *abstract non-interference* [10] and abstract slicing. A formalization of this relationships may be one more object of the future work, and may lead to an implementation of abstract non-interference certifications for program security.

References

1. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B.: A formalisation of the relationship between forms of program slicing. Sci. Comput. Program 62(3), 228–252 (2006)
2. Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., Korel, B.: Theoretical foundations of dynamic program slicing. Theor. Comput. Sci. 360(1), 23–41 (2006)
3. Binkley, D.W., Gallagher, K.B.: Program slicing. Advances in Computers 43 (1996)
4. Canfora, G., Cinitile, A., De Lucia, A.: Conditioned program slicing. Information and Software Tech. 40, 11–12 (1998)
5. Cimitile, A., De Lucia, A., Munro, M.: A specification driven slicing process for identifying reusable functions. Journal of Software Maintenance 8(3), 145–178 (1996)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of Conf. Record of the 4th ACM Symp. on Principles of Programming Languages (POPL 1977), pp. 238–252. ACM Press, New York (1977)
7. De Lucia, A.: Program slicing: Methods and applications. In: IEEE International Workshop on Source Code Analysis and Manipulation (2001)
8. Field, J., Ramalingam, G., Tip, F.: Parametric program slicing. In: POPL 1995: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 379–392. ACM, New York (1995)
9. Gallagher, K.B., Lyle, J.R.: Using program slicing in software maintenance. IEEE Trans. on Software Engineering 17(8), 751–761 (1991)
10. Giacobazzi, R., Mastroeni, I.: Abstract non-interference: Parameterizing non-interference by abstract interpretation. In: Proc. of the 31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2004), pp. 186–197. ACM-Press, New York (2004)
11. Hong, H.S., Lee, I., Sokolsky, O.: Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In: Proc. of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2005), pp. 25–34. IEEE Comp. Soc. Press, Los Alamitos (2005)
12. Korel, B., Laski, J.: Dynamic program slicing. Information Processing Letters 29(3), 155–183 (1988)
13. Mastroeni, I., Zanardini, D.: Data dependencies and program slicing: From syntax to abstract semantics. In: Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-Based Program Manipulation (PEPM 2008), pp. 125–134 (2008)
14. Tip, F.: A survey of program slicing techniques. J. of Programming Languages 3, 121–189 (1995)
15. Weiser, M.: Program slicing. IEEE Trans. on Software Engineering 10(4), 352–357 (1984)
16. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: better together! In: ISSTA 2006: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp. 145–156. ACM, New York (2006)

Loop Invariant Synthesis in a Combined Domain

Shengchao Qin[1], Guanhua He[2], Chenguang Luo[2,⋆], and Wei-Ngan Chin[3]

[1] Teesside University, Middlesbrough, TS1 3BA, UK
[2] Durham University, Durham, DH1 3LE, UK
[3] National University of Singapore

Abstract. Automated verification of memory safety and functional correctness for heap-manipulating programs has been a challenging task, especially when dealing with complex data structures with strong invariants involving both shape and numerical properties. Existing verification systems usually rely on users to supply annotations, which can be tedious and error-prone and can significantly restrict the scalability of the verification system. In this paper, we reduce the need of user annotations by automatically inferring loop invariants over an abstract domain with both separation and numerical information. Our loop invariant synthesis is conducted automatically by a fixpoint iteration process, equipped with newly designed abstraction mechanism, and join and widening operators. Initial experiments have confirmed that we can synthesise loop invariants with non-trivial constraints.

1 Introduction

Although research on software verification has a long and distinguished history dating back to the 1960's, it remains a challenging problem to automatically verify heap manipulating programs written in mainstream imperative languages. This is in part due to the shared mutable data structures lying in programs, and the need to track various program properties, such as structural numerical information (length and height) and relational numerical information (sortedness and binary search tree properties).

Since the emergence of separation logic [14,25], dramatic advances have been made in automated software verification, e.g. the Smallfoot tool [1] for the verification on pointer safety (i.e. shape properties asserting that pointers cannot go wrong), the verification on termination [2], the verification for object-oriented programs [6,22], and Dafny [18] and Hip/Sleek [5,20,21] for more general properties (both structural and numerical ones) for heap-manipulating programs.

These verification systems generally require users to provide specifications for each method as well as invariants for each loop, which is both tedious and error-prone. This also affects their scalability, as there can be many methods in a program and each method may still contain several while loops.

To conquer this problem, separation logic based shape analysis techniques are brought in, e.g., the SpaceInvader tool [3,9,27]. As a further step of Smallfoot, it automatically infers method specifications and loop invariants for pointer

⋆ Now with Citigroup Inc.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 468–484, 2010.
© Springer-Verlag Berlin Heidelberg 2010

safety in the shape domain. Other works such as THOR [19] incorporate simple numerical information into the shape domain to allow automated synthesis of properties like list length. Their success proves the necessity and feasibility for shape analysis to help automate the verification process.

However, the prior analyses focus mainly on relatively simple properties, such as pointer safety for lists and list length information. It is difficult to apply them in the presence of more sophisticated program properties, such as:

– More flexible user-defined data structures, such as trees;
– Relational numerical properties, like sortedness and binary search property.

These properties can be part of the full functional correctness of heap-manipulating programs. The (aforementioned) Hip/Sleek tool aims to verify such properties and it allows users to define their own shape predicates to express their desired level of correctness.

In this paper, we make the first stride to improve the level of automation for Hip/Sleek-like verification systems by discovering loop invariants automatically over the combined shape and numerical domain. This proves to be a challenging problem especially since we aim towards full functional correctness that Hip/Sleek targets at. Our approach is based on the framework of abstract interpretation [7] with fixpoint computation. It makes the following contributions in summary:

– We propose a loop invariant synthesis with novel operations for abstraction, join and widening over a combined shape and numerical domain.
– We demonstrate that our analysis is sound w.r.t. concrete program semantics and always terminates.
– We have integrated our solution with Hip/Sleek and conducted some initial experiments. The experimental results confirm the viability of our solution and show that we can effectively eliminate the need for user-provision of loop invariants which were previously necessary in verification.

We shall next illustrate our approach informally via an example before presenting the formal details.

2 The Approach

Before giving an illustrative example for the analysis, we will first introduce our specification mechanism which follows the Hip/Sleek system.

2.1 Separation Logic and User-Defined Predicates

Separation logic [14,25] extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic: separation conjunction $*$ and spatial implication $-*$. The formula $\mathtt{p}_1 * \mathtt{p}_2$ asserts that two heaps described by the formulae $\mathtt{p}_1$ and $\mathtt{p}_2$ are domain-disjoint, while $\mathtt{p}_1 -* \mathtt{p}_2$ asserts that if the current heap is extended with a disjoint heap described by the formula $\mathtt{p}_1$, then the formula $\mathtt{p}_2$ holds in the extended heap. In this paper we only use separation conjunction.

Similar to the Hip/Sleek system, we allow user-defined inductive predicates to specify both separation and numerical properties. For example, with a data structure definition for a node in a list `data node { int val; node next; }`, we can define a predicate for a list as

$$\texttt{root::ll}\langle\texttt{n}\rangle \equiv (\texttt{root=null}\wedge\texttt{n=0})\vee(\exists \texttt{v},\texttt{q},\texttt{m}\cdot\texttt{root::node}\langle\texttt{v},\texttt{q}\rangle{*}\texttt{q::ll}\langle\texttt{m}\rangle\wedge\texttt{n=m+1})$$

The parameter `root` for the predicate `ll` is the root pointer referring to the list. Its length is denoted by `n`. A uniform notation $\texttt{p::c}\langle\texttt{v}_1,..,\texttt{v}_k\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node with fields $\texttt{f}_1,..,\texttt{f}_k$, the notation represents a singleton heap, $\texttt{p}{\mapsto}\texttt{c}[\texttt{f}_1{:}\texttt{v}_1,..,\texttt{f}_k{:}\texttt{v}_k]$, e.g. the $\texttt{root::node}\langle\texttt{v},\texttt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\texttt{v}_1,..,\texttt{v}_k$, e.g., the $\texttt{q::ll}\langle\texttt{m}\rangle$ above.

We can also define a list segment as follows:

$$\texttt{ls}\langle\texttt{p},\texttt{n}\rangle \equiv (\texttt{root=p}\wedge\texttt{n=0}) \vee (\texttt{root::node}\langle_,\texttt{q}\rangle * \texttt{q::ls}\langle\texttt{p},\texttt{m}\rangle \wedge \texttt{n=m+1})$$

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and `m`, are implicitly existentially quantified, and (iii) the underscore _ denotes an existentially quantified anonymous variable.

If the user wants to verify a sorting algorithm, they can incorporate sortedness property into the above predicates as follows:

$$\begin{aligned}
&\texttt{sll}\langle\texttt{n},\texttt{mn},\texttt{mx}\rangle \equiv (\texttt{root=null}\wedge\texttt{n=0}\wedge\texttt{mn=mx}) \vee \\
&\quad(\texttt{root::node}\langle\texttt{mn},\texttt{q}\rangle * \texttt{q::sll}\langle\texttt{n}_1,\texttt{k},\texttt{mx}\rangle \wedge \texttt{mn}{\le}\texttt{k} \wedge \texttt{n=n}_1\texttt{+1})\\
&\texttt{sls}\langle\texttt{p},\texttt{n},\texttt{mn},\texttt{mx}\rangle \equiv (\texttt{root=p}\wedge\texttt{n=0}\wedge\texttt{mn=mx}) \vee \\
&\quad(\texttt{root::node}\langle\texttt{mn},\texttt{q}\rangle * \texttt{q::sls}\langle\texttt{p},\texttt{n}_1,\texttt{k},\texttt{mx}\rangle \wedge \texttt{mn}{\le}\texttt{k} \wedge \texttt{n=n}_1\texttt{+1})
\end{aligned}$$

where `mn` and `mx` denote resp. the min and max values stored in the sorted list. Such user-supplied predicates can be used to specify loop invariants and method pre/post-specifications.

2.2 Illustrative Example

We now illustrate via an example our loop invariant synthesis process. The method `ins_sort` (Figure 1) sorts a linked list with the insertion sort algorithm. It is implemented with two nested while loops. The outer loop traverses the whole list `x`, takes out each node from it (line 7), and inserts that node into another already sorted list `r` (which is empty initially before the sorting). This insertion process makes use of the inner while loop in lines 9-11 to look for a proper position in the already sorted list for the new node to be inserted. The actual insertion takes place at lines 12-14.

To verify this program, we need to synthesise appropriate loop invariants for both while loops. Our analysis follows a standard fixpoint iteration process. It starts with the (abstract) program state immediately before the while loop (i.e., the initial state) and symbolically executes the loop body for several iterations,

```
0 data node { int val;                  9   while (srt != null &&
              node next; }                        srt.val <= v) {
1 node ins_sort(node x)                10     prv=srt; srt=srt.next;
2  requires x::ll⟨n⟩                   11   }
3  ensures res::sll⟨n,mn,mx⟩           12   cur.next=srt;
4 {int v;                              13   if (prv != null) prv.next=cur;
5  node r,cur,srt,prv=null;            14   else r=cur;
6  while (x != null) {                 15  }
7   cur=x; x=x.next; v=cur.val;        16  return r;
8   srt=r; prv=null;                   17 }
```

Fig. 1. Insertion sort for linked list

until the obtained states converge to a fixpoint, which is the loop invariant.[1] At the start of each iteration, the obtained state from the previous iteration is joined with the initial state. In addition to this join operator, we have also defined an abstraction function and a widening operator both of which will help the fixpoint iteration to converge. The join and widening operators are specifically designed to handle both shape and numerical information.

As for our example, due to the presence of nested loops, each iteration of the analysis for the outer loop actually infers a loop invariant for the inner loop. We shall now illustrate how we synthesise a loop invariant for the inner loop.

Suppose that in one iteration for the outer loop, the state at line 9 becomes

$$\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle * \mathtt{cur}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n}$$

Note that since the inner loop does not mutate the heap part referred to by `cur` and `x` (i.e., $\mathtt{cur}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_x}\rangle$), we can ignore it during the invariant synthesis and add it back to the program state using the frame rule of separation logic [25]. Therefore, the initial state for loop invariant synthesis becomes

$$\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n} \tag{1}$$

From this state, symbolically executing the loop body once yields the state:

$$\begin{array}{c}\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{r} \wedge \\ \mathtt{a}{\le}\mathtt{c_1} \wedge \mathtt{a}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r}\end{array} \tag{2}$$

which says that pointer `srt` moves towards the tail of the list for one node. Then we join it with the initial state (1) to obtain

$$\begin{array}{l}(\mathtt{r}{::}\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n})\vee \\ (\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \\ \qquad \mathtt{prv}{=}\mathtt{r} \wedge \mathtt{a}{\le}\mathtt{c_1} \wedge \mathtt{a}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r})\end{array} \tag{3}$$

The second iteration over the loop body starts with (3) and exhibits (also) the case that `srt` runs two nodes towards tail, while `prv` goes one node. Its result is then joined with pre-state (1) to become the current state:

[1] The fixpoint iteration converges if one more iteration still yields the same result.

$$(3) \vee \mathtt{r{::}node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv{::}node}\langle \mathtt{c_1}, \mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s}, \mathtt{c_2}, \mathtt{b}\rangle \wedge \mathtt{a}{\le}\mathtt{c_1}{\le}\mathtt{c_2} \wedge \mathtt{c_1}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}2{=}\mathtt{n_r} \quad (4)$$

Executing the loop body a third time returns a post-state where three nodes are passed by srt, and two by prv, as below:

$$(4) \vee \mathtt{r{::}node}\langle \mathtt{a}, \mathtt{r_0}\rangle * \mathtt{r_0{::}node}\langle \mathtt{c_1}, \mathtt{prv}\rangle * \mathtt{prv{::}node}\langle \mathtt{c_2}, \mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \mathtt{a}{\le}\mathtt{c_1}{\le}\mathtt{c_2}{\le}\mathtt{c_3} \wedge \mathtt{c_2}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}3{=}\mathtt{n_r}$$

where we have an auxiliary logical variable $\mathtt{r_0}$. Following this trend, it is predictable that every iteration hereafter will introduce an additional logical variable (referring to a list node). If we indulge in such increase in the subsequent iterations, the analysis will never terminate. Our abstraction process prevents this from happening by eliminating such logical variables as follows:

$$(4) \vee \mathtt{r{::}sls}\langle \mathtt{prv}, \mathtt{n_1}, \mathtt{a}, \mathtt{c_1}\rangle * \mathtt{prv{::}node}\langle \mathtt{c_2}, \mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \mathtt{a}{\le}\mathtt{c_1}{\le}\mathtt{c_2}{\le}\mathtt{c_3} \wedge \mathtt{c_2}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}3{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}2$$

Note that the heap part $\mathtt{r{::}node}\langle \mathtt{a}, \mathtt{r_0}\rangle * \mathtt{r_0{::}node}\langle \mathtt{c_1}, \mathtt{prv}\rangle$ is abstracted as a sorted list segment $\mathtt{r{::}sls}\langle \mathtt{prv}, \mathtt{n_1}, \mathtt{a}, \mathtt{c_1}\rangle$ with $\mathtt{n_1}$ denoting the length of the segment and $\mathtt{n_1}{=}2$ added into the state. This abstraction process ensures that our analysis does not allow the shape to increase infinitely.

The fourth iteration responds with a post-state where four nodes are passed by srt, and three by prv. Therefore an abstraction is performed to remove the logical pointer variables. As a simplification of the presentation, we denote σ as $\mathtt{r{::}sls}\langle \mathtt{prv},\mathtt{n_1},\mathtt{a},\mathtt{c_1}\rangle * \mathtt{prv{::}node}\langle \mathtt{c_2},\mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s},\mathtt{c_3},\mathtt{b}\rangle \wedge \mathtt{a}{\le}\mathtt{c_1}{\le}\mathtt{c_2}{\le}\mathtt{c_3} \wedge \mathtt{c_2}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x}$, and the abstracted result (after the fourth iteration) is

$$(4) \vee (\sigma \wedge \mathtt{n_s}{+}3{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}2) \vee (\sigma \wedge \mathtt{n_s}{+}4{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}3)$$

for which we have an observation that the last two disjunctions share the same shape part (as in σ). This disjunction will be transferred to the numerical domain, as follows:

$$(4) \vee (\sigma \wedge (\mathtt{n_s}{+}3{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}2 \vee \mathtt{n_s}{+}4{=}\mathtt{n_r} \wedge \mathtt{n_1}{=}3))$$

This simplifies the abstraction further. After that, our widening operation compares the current state with the previous one, to look for the same (numerical) constraints that both states imply, and to replace those numerical constraints in the current state with the ones discovered by widening. This operation eventually ensures termination of our analysis. As for the example, some constraints among $\mathtt{n_s}$, $\mathtt{n_r}$ and $\mathtt{n_1}$ can be found to make the widened post-state become:

$$(4) \vee (\sigma \wedge \mathtt{n_s}{+}\mathtt{n_1}{=}\mathtt{n_r}{-}1 \wedge \mathtt{n_1}{\ge}2) \quad (5)$$

One more iteration of symbolic execution will produce the same result as (5), suggesting that it is already the fixpoint (and hence the loop invariant):

$$\begin{array}{l}
\mathtt{r{::}sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \vee \\
\mathtt{r{::}node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{r} \wedge \\
\quad \mathtt{a}{\le}\mathtt{c_1} \wedge \mathtt{a}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r} \vee \\
\mathtt{r{::}node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv{::}node}\langle \mathtt{c_1}, \mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s}, \mathtt{c_2}, \mathtt{b}\rangle \wedge \\
\quad \mathtt{a}{\le}\mathtt{c_1}{\le}\mathtt{c_2} \wedge \mathtt{c_1}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}2{=}\mathtt{n_r} \vee \\
\mathtt{r{::}sls}\langle \mathtt{prv}, \mathtt{n_1}, \mathtt{a}, \mathtt{c_1}\rangle * \mathtt{prv{::}node}\langle \mathtt{c_2}, \mathtt{srt}\rangle * \mathtt{srt{::}sll}\langle \mathtt{n_s}, \mathtt{c_3}, \mathtt{b}\rangle \wedge \\
\quad \mathtt{a}{\le}\mathtt{c_1}{\le}\mathtt{c_2}{\le}\mathtt{c_3} \wedge \mathtt{c_2}{\le}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}\mathtt{n_1}{=}\mathtt{n_r}{-}1 \wedge \mathtt{n_1}{\ge}2
\end{array}$$

Note that although it is possible to further join the third disjunctive branch with the fourth, our analysis does not do so as it tries to keep the result as precise as possible by eliminating only auxiliary pointer variables.

With the frame part $\mathtt{cur}::\mathtt{node}\langle\mathtt{v},\mathtt{x}\rangle * \mathtt{x}::\mathtt{ll}\langle\mathtt{n_x}\rangle$ added back, the analysis for the outer loop continues. Eventually, the following loop invariant is discovered for the outer loop:

$$(\mathtt{x}::\mathtt{ll}\langle\mathtt{n_x}\rangle \wedge \mathtt{r}{=}\mathtt{null} \wedge \mathtt{n_x}{=}\mathtt{n}) \vee (\mathtt{r}::\mathtt{node}\langle\mathtt{a},\mathtt{null}\rangle * \mathtt{x}::\mathtt{ll}\langle\mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}1) \vee \\ (\mathtt{r}::\mathtt{sll}\langle\mathtt{n_r},\mathtt{a},\mathtt{b}\rangle * \mathtt{x}::\mathtt{ll}\langle\mathtt{n_x}\rangle \wedge \mathtt{n}{=}\mathtt{n_x}{+}\mathtt{n_r} \wedge \mathtt{n_r}{\geq}2)$$

which allows us to verify the whole method successfully using e.g. Hip/Sleek.

3 Language and Abstract Domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Figure 2. The program *Prog* written in this language consists of declarations *tdecl*, which can either be data type declarations *datat* (e.g. `node` in Section 2), or predicate definitions *spred* (e.g. `ll`, `ls`, `sll`, `sls` in Section 2.1), as well as method declarations *meth*. The definitions for *spred* and *mspec* are given later in Figure 3. Without loss of expressiveness, we use an expression-oriented language. So the body of a method (e) is an expression formed by standard commands of an imperative language. Note that d and $d[v]$ represent resp. heap-insensitive and heap sensitive commands. The language allows both call-by-value and call-by-reference method parameters (separated with a semicolon ;).

$$
\begin{array}{ll}
\mathit{Prog} ::= \mathit{tdecl}^*\ \mathit{meth}^* & \mathit{tdecl} ::= \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} ::= \mathtt{data}\ c\ \{\ \mathit{field}^*\ \} & \mathit{field} ::= t\ v \qquad t ::= c \mid \tau \\
\mathit{meth} ::= t\ mn\ ((t\ v)^*; (t\ v)^*)\ \mathit{mspec}\ \{e\} & \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{void} \\
e \quad ::= d \mid d[v] \mid v{:=}e \mid e_1; e_2 \mid t\ v;\ e \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{while}\ v\ \{e\} & \\
d \quad ::= \mathtt{null} \mid k^\tau \mid v \mid \mathtt{new}\ c(v^*) \mid mn(u^*; v^*) & \\
d[v] \ ::= v.f \mid v.f{:=}w \mid \mathtt{free}(v) &
\end{array}
$$

Fig. 2. A Core (C-like) Imperative Language

Our specification language (in Figure 3) allows (user-defined) shape predicates *spred* to specify both shape and numerical properties. Note that *spred* are constructed with disjunctive constraints Φ and numerical formulae π. We require that the predicates be well-formed [21].

A conjunctive abstract program state, σ, is composed of a heap (shape) part κ and a numerical part π, where π consists of γ and ϕ as aliasing and numerical information, respectively. We use SH to denote the set of such conjunctive states. During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ (and its set is recognised as $\mathcal{P}_{\mathsf{SH}}$). An abstract state Δ can be normalised to the Φ form.

Using entailment [21], we define a partial order over these abstract states:

$$\Delta \preceq \Delta' =_{df} \Delta' \vdash \Delta * \mathtt{R}$$

$$
\begin{array}{lll}
spred ::= \texttt{root}::c\langle v^*\rangle \equiv \Phi & \Phi ::= \bigvee \sigma^* & \sigma ::= \exists v^* \cdot \kappa \wedge \pi \\
mspec ::= requires\ \Phi_{pr}\ ensures\ \Phi_{po} & & \\
\Delta \quad ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta & & \\
\kappa \quad ::= \texttt{emp} \mid v::c\langle v^*\rangle \mid \kappa_1 * \kappa_2 & & \pi ::= \gamma \wedge \phi \\
\gamma \quad ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \texttt{true} \mid \gamma_1 \wedge \gamma_2 & & \\
\phi \quad ::= b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & & \\
b \quad ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 & & a ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s \quad ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times s \mid s_1 + s_2 \mid -s \mid max(s_1,s_2) \mid min(s_1,s_2) & &
\end{array}
$$

Fig. 3. The Specification Language

where R is the (computed) residue part. And we also have an induced lattice over these states as the base of fixpoint calculation for loop invariants.

The memory model of our specification formula is similar to the model given for separation logic [25], except that we have extensions to handle user-defined shape predicates and related numerical properties. In our analysis, all the variables except the program ones are logical variables. We denote a program variable's initial value as unprimed and its current value as primed [21].

4 Analysis Algorithm

Our proposed analysis algorithm is given in Figure 4.

Fixpoint Computation in Combined Domain
Input: $\mathcal{T}$, Δ_{pre}, `while` b $\{e\}$, n ;
Local: $i := 0$; $\Delta_i :=$ `false`; $\Delta'_i :=$ `false`;
1 **repeat**
2 $i := i + 1$;
3 $\Delta_i := \mathsf{widen}^\dagger(\Delta_{i-1}, \mathsf{join}^\dagger(\Delta_{pre}, \Delta'_{i-1}))$;
4 $\Delta'_i := \mathsf{abs}^\dagger([\![e]\!]_{\mathcal{T}}(\Delta_i \wedge b))$;
5 **if** $\Delta'_i =$ `false` $\vee$ $\mathsf{cp_no}(\Delta'_i) > n$
then return fail **end if**
6 **until** $\Delta'_i = \Delta'_{i-1}$;
7 **return** Δ'_i

Fig. 4. Main analysis algorithm

The algorithm takes four input parameters: $\mathcal{T}$ as the program environment with all the method specifications in the program (for potential method calls in loop body), Δ_{pre} as the precondition of the loop's (symbolic) execution, the while loop itself `while` b $\{e\}$, and the number of upper bound of shared logical variables we keep during the analysis n.

Our analysis is based on abstract interpretation [7] with specifically designed operations (abs, join and widen) over this combined domain.[2] At the beginning, we initialise the iteration variable (i) and two states to begin with (Δ_i and Δ'_i). The `false`'s here as initial values denote the top element of our defined lattice

[2] Note that our analysis uses lifted versions of these operations (indicated by †), which will be explained in more details in Section 4.2.

as well as our starting point of the fixpoint iteration. Among the two states here, the unprimed version Δ_i denotes the initial state before the ith execution of the loop body, and the primed one Δ'_i represents the result state after. Each iteration starts at line 1. Firstly we join together the precondition of the loop with the result state Δ'_{i-1} obtained in the previous iteration, and widen it against the initial state Δ_{i-1} of the previous iteration (line 3). Then we symbolically execute the loop body with the abstract semantics in Section 4.1 (line 4), and apply the abstraction operation to the obtained abstract state. If the symbolic execution cannot continue due to a program bug, or if we find our abstraction cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp_no}$) within a specified bound (n), then a failure is reported (line 5). Otherwise we judge whether a fixpoint is already reached by comparing the current abstract state with the previous one (line 6). The fixpoint Δ'_i is returned as the loop invariant.

We will elaborate the key techniques of our analysis in what follows: the abstract semantics, the abstraction function, and the join and widening operators.

4.1 Abstract Semantics

The abstract semantics is used to execute the loop body symbolically to obtain its post-state during the loop invariant synthesis. Its type is defined as

$$[\![e]\!] \;:\; \mathsf{AllSpec} \rightarrow \mathcal{P}_{\mathsf{SH}} \rightarrow \mathcal{P}_{\mathsf{SH}}$$

where $\mathsf{AllSpec}$ contains all the specifications of all methods (extracted from the program *Prog*). For some expression e, given its precondition, the semantics will calculate the postcondition.

The foundation of the semantics is the basic transition functions from a conjunctive abstract state to a conjunctive or disjunctive abstract state below:

$$\begin{array}{lll} \mathsf{rearr}(x) & : \; \mathsf{SH} \rightarrow \mathcal{P}_{\mathsf{SH}[x]} & \text{Rearrangement} \\ \mathsf{exec}(d[x]) & : \; \mathsf{AllSpec} \rightarrow \mathsf{SH}[x] \rightarrow \mathsf{SH} & \text{Heap-sensitive execution} \\ \mathsf{exec}(d) & : \; \mathsf{AllSpec} \rightarrow \mathsf{SH} \rightarrow \mathsf{SH} & \text{Heap-insensitive execution} \end{array}$$

where $\mathsf{SH}[x]$ denotes the set of conjunctive abstract states in which each element has x exposed as the head of a data node ($x{::}c\langle v^*\rangle$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\mathsf{rearr}(x)$ rearranges the symbolic heap so that the cell referred to by x is exposed for access by heap sensitive commands $d[x]$ via the second transition function $\mathsf{exec}(d[x])$. The third function defined for other (heap insensitive) commands d does not require such exposure of x.

$$\frac{\mathit{isdatat}(c) \quad \sigma \vdash x{::}c\langle v^*\rangle * \sigma'}{\mathsf{rearr}(x)\sigma =_{df} \sigma} \qquad \frac{\mathit{isspred}(c) \quad \sigma \vdash x{::}c\langle u^*\rangle * \sigma' \quad \mathtt{root}{::}c\langle v^*\rangle {\equiv} \Phi}{\mathsf{rearr}(x)\sigma =_{df} \sigma' * [x/\mathtt{root}, u^*/v^*]\Phi}$$

The test $\mathit{isdatat}(c)$ returns $\mathtt{true}$ only if c is a data node and $\mathit{isspred}(c)$ returns $\mathtt{true}$ only if c is a shape predicate.

The symbolic execution of heap-sensitive commands $d[x]$ (i.e. $x.f_i$, $x.f_i := w$, or $\mathtt{free}(x)$) assumes that the rearrangement $\mathsf{rearr}(x)$ has been done in prior:

$$\frac{isdatat(c) \qquad \sigma \vdash x{::}c\langle v_1, .., v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})\sigma =_{df} \sigma' * x{::}c\langle v_1, .., v_n\rangle \wedge \mathtt{res}{=}v_i}$$

$$\frac{isdatat(c) \qquad \sigma \vdash x{::}c\langle v_1, .., v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})\sigma =_{df} \sigma' * x{::}c\langle v_1, .., v_{i-1}, w, v_{i+1}, .., v_n\rangle}$$

$$\frac{isdatat(c) \qquad \sigma \vdash x{::}c\langle u^*\rangle * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})\sigma =_{df} \sigma'}$$

The symbolic execution rules for heap-insensitive commands are as follows:

$$\mathsf{exec}(k)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}k \qquad \mathsf{exec}(x)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}x$$

$$\frac{isdatat(c)}{\mathsf{exec}(\mathtt{new}\ c(v^*))(\mathcal{T})\sigma =_{df} \sigma * \mathtt{res}{::}c\langle v^*\rangle}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t'_i\ v_i)_{i=1}^n)\ requires\ \Phi_{pr}\ ensures\ \Phi_{po} \in \mathcal{T} \\ \rho = [x'_i/u_i]_{i=1}^m \circ [y'_i/v_i]_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma' \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x'_i/u'_i]_{i=1}^m \circ [y'_i/v'_i]_{i=1}^n \quad \rho_l = [r_i/y'_i]_{i=1}^n \quad fresh\ logical\ r_i \end{array}}{\mathsf{exec}(mn(x_1, .., x_m; y_1, .., y_n))(\mathcal{T})\sigma =_{df} (\rho_l\sigma') * (\rho_o\Phi_{po})}$$

Note that the first three rules deal with constant (k), variable (x) and data node creation ($\mathtt{new}\ c(v^*)$), respectively, while the last rule handles method invocation. In the last rule, the call site is ensured to meet the precondition of mn, as signified by $\sigma \vdash \rho\Phi_{pr} * \sigma'$. In this case, the execution succeeds and the post-state of the method call involves mn's postcondition as signified by $\rho_{ol} \circ \rho_o\Phi_{po}$.

A lifting function † is defined to lift rearr's domain to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{rearr}^{\dagger}(x) \bigvee \sigma_i =_{df} \bigvee(\mathsf{rearr}(x)\sigma_i)$$

and this function is overloaded for exec to lift both its domain and range to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{exec}^{\dagger}(d)(\mathcal{T}) \bigvee \sigma_i =_{df} \bigvee(\mathsf{exec}(d)(\mathcal{T})\sigma_i)$$

Based on the transition functions above, we can define the abstract semantics for a program command e as follows:

$$\begin{array}{lll} [\![d[x]]\!]_{\mathcal{T}}\Delta & =_{df} & \mathsf{exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{rearr}^{\dagger}(x)\Delta \\ [\![d]\!]_{\mathcal{T}}\Delta & =_{df} & \mathsf{exec}^{\dagger}(d)(\mathcal{T})\Delta \\ [\![e_1; e_2]\!]_{\mathcal{T}}\Delta & =_{df} & [\![e_2]\!]_{\mathcal{T}} \circ [\![e_1]\!]_{\mathcal{T}}\Delta \\ [\![x := e]\!]_{\mathcal{T}}\Delta & =_{df} & [x'/x, r'/\mathtt{res}]([\![e]\!]_{\mathcal{T}}\Delta) \wedge x{=}r' \quad fresh\ logical\ x', r' \\ [\![\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}\Delta & =_{df} & ([\![e_1]\!]_{\mathcal{T}}(v{\wedge}\Delta)) \vee ([\![e_2]\!]_{\mathcal{T}}(\neg v{\wedge}\Delta)) \end{array}$$

which form the foundation for us to analyse the loop body.

4.2 Abstraction, Join and Widening

This section describes our specifically designed abstraction, join and widening operations employed in our loop invariant synthesis process.

Abstraction function. During the symbolic execution, we may be confronted with many "concrete" shapes in postconditions of the loop body. As an example

of list traversal, the list may contain one node, or two nodes, or even more nodes in the list, which the analysis cannot enumerate infinitely. The abstraction function deals with those situations by abstracting the (potentially infinite) concrete situations into more abstract shapes. Our rationale is to keep only program variables and shared cutpoints; all other logical variables will be abstracted away. As an instance, the first state below can be further abstracted (as shown), while the second one cannot:

$$\begin{array}{l} \mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle_,\mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle_,\mathtt{null}\rangle) \;=\; \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{=}2 \\ \mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle_,\mathtt{z}_0\rangle * \mathtt{y}{::}\mathtt{node}\langle_,\mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle_,\mathtt{null}\rangle) \;=\; \text{-} \end{array} \quad (6)$$

where both $\mathtt{x}$ and $\mathtt{y}$ are program variables, and $\mathtt{z}_0$ is an existentially quantified logical variable. In the second case $\mathtt{z}_0$ is a shared cutpoint referenced by both $\mathtt{x}$ and $\mathtt{y}$, and thus the state is not changed. As illustrated, the abstraction transition function abs eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

$$\mathsf{abs} \;:\; \mathsf{SH} \rightarrow \mathsf{SH} \qquad \text{Abstraction}$$

which indicates that it takes in a conjunctive abstract state σ and abstracts it as another conjunctive state σ'. Below are its rules.

$$\mathsf{abs}(\sigma \wedge x_0{=}e) =_{df} \sigma[e/x_0] \qquad \mathsf{abs}(\sigma \wedge e{=}x_0) =_{df} \sigma[e/x_0]$$

$$\frac{\neg\mathsf{Reach}(\sigma, x_0)}{\mathsf{abs}(x_0{::}c\langle v^*\rangle * \sigma) =_{df} \sigma * \mathtt{true}}$$

$$\frac{\begin{array}{c} \mathit{isdatat}(c_1) \qquad c_2\langle u_2^*\rangle \equiv \Phi \\ p{::}c_1\langle v_1^*\rangle * \sigma_1 \vdash p{::}c_2\langle v_2^*\rangle * \sigma_2 \qquad \neg\mathsf{Reach}(p{::}c_2\langle v_2^*\rangle * \sigma_2, v_1^*) \end{array}}{\mathsf{abs}(p{::}c_1\langle v_1^*\rangle * \sigma_1) =_{df} p{::}c_2\langle v_2^*\rangle * \sigma_2}$$

The first two rules eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule is used to eliminate any garbage (heap part led by a logical variable x_0 unreachable from the other part of the heap) that may exist in the heap. As x_0 is already unreachable from, and not usable by, the program variables, it is safe to treat it as garbage $\mathtt{true}$, for example the $\mathtt{x}_0$ in $\mathtt{x}{::}\mathtt{node}\langle_,\mathtt{null}\rangle * \mathtt{x}_0{::}\mathtt{node}\langle_,\mathtt{null}\rangle$ where only $\mathtt{x}$ is a program variable.

The last rule of abs plays the most significant role which intends to eliminate shape formulae led by logical variables (all variables in v_1^*). It tries to fold data nodes up to a predicate node. It confirms that c_1 is a data node definition and c_2 is a predicate. Meanwhile it also ensures that the latter is a sound abstraction of the former by entailment checking, and the logical parameters of c_1 are not reachable from other part of the heap (so that the abstraction does not lose necessary information). The predicate Reach is defined as follows:

$$\mathsf{Reach}(\sigma, x^*) =_{df} \{x^*\} \subseteq \bigcup_{v \in \mathtt{fv}(\sigma)} \mathtt{ReachVar}(\kappa{\wedge}\pi, v) \;\mathtt{where}\; \sigma ::= \exists \mathtt{u}^*{\cdot}\kappa{\wedge}\pi$$

saying that each variable in x^* is reachable from some free variable in the abstract state σ. The function $\mathtt{ReachVar}(\kappa \wedge \pi, v)$ returns the minimal set of variables satisfying the relationship below:

$$\{v\} \cup \{z_2 \mid \exists z_1, \pi_1 \cdot z_1 \in \texttt{ReachVar}(\kappa \wedge \pi, v) \wedge \pi{=}(z_1{=}z_2 \wedge \pi_1)\} \cup \{z_2 \mid \exists z_1, \kappa_1 \cdot z_1 \in \texttt{ReachVar}(\kappa \wedge \pi, v) \wedge \kappa{=}(z_1{::}c\langle .., z_2, ..\rangle * \kappa_1)\} \subseteq \texttt{ReachVar}(\kappa \wedge \pi, v)$$

That is, it is composed of aliases of v and variables reachable from v. As in the previous example: $\mathsf{abs}(\texttt{x::node}\langle _, \texttt{z}_0\rangle * \texttt{z}_0\texttt{::node}\langle _, \texttt{null}\rangle) \rightsquigarrow \texttt{x::ll}\langle \texttt{n}_0\rangle \wedge \texttt{n}_0{=}2$.

During the analysis, we apply the above abstraction rules (following the given order) onto the current abstract state exhaustively until it stabilises. Such convergence is confirmed because the abstract shape domain is finite due to the bounded numbers of variables and predicates, as discussed later.

Finally the lifting function is overloaded for abs to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{abs}^{\dagger} \bigvee \sigma_i =_{df} \bigvee \mathsf{abs}(\sigma_i)$$

which allows it to be used in the analysis.

Join operator. The operator join is applied over two conjunctive abstract states, trying to find a common shape as a sound abstraction for both:

$$\begin{array}{l} \mathsf{join}(\sigma_1, \sigma_2) =_{df} \\ \quad \textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in} \\ \quad\quad \textbf{match } \sigma_1', \sigma_2' \textbf{ with } (\exists x_1^* \cdot \kappa_1 \wedge \pi_1), (\exists x_2^* \cdot \kappa_2 \wedge \pi_2) \textbf{ in} \\ \quad\quad\quad \textbf{if } \kappa_1 \vdash \kappa_2 * \texttt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\ \quad\quad\quad \textbf{else if } \kappa_2 \vdash \kappa_1 * \texttt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_1 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2)) \\ \quad\quad\quad \textbf{else } \sigma_1 \vee \sigma_2 \end{array}$$

where the rename function prevents naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of same name in the two states with fresh names. For example it will renew $\texttt{x}_0$'s name in both states $\exists \texttt{x}_0 \cdot \texttt{x}_0{=}0$ and $\exists \texttt{x}_0 \cdot \texttt{x}_0{=}1$ to make them $\exists \texttt{x}_0 \cdot \texttt{x}_0{=}0$ and $\exists \texttt{x}_1 \cdot \texttt{x}_1{=}1$. After this procedure it judges whether σ_2 is an abstraction of σ_1, or the other way round. If either case holds, it regards the shape of the weaker state as the shape of the joined states, and performs joining for numerical formulae with $\mathsf{join}_\pi(\pi_1, \pi_2)$, the convex hull operator over numerical domain [23]. Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case). Then we lift this operator for abstract state Δ as follows:

$$\mathsf{join}^{\dagger}(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\bigvee_i \sigma_i^1), (\bigvee_j \sigma_j^2) \textbf{ in } \bigvee_{i,j} \mathsf{join}(\sigma_i^1, \sigma_j^2)$$

which essentially joins all pairs of disjunctions from the two abstract states, and makes a disjunction of them.

Widening operator. The finiteness of the shape domain is confirmed by the abstraction function. To ensure the termination of the whole analysis, we still need to guarantee the convergence over the numerical domain. This task is accomplished by the widening operator.

The widening operator $\mathsf{widen}(\sigma_1, \sigma_2)$ is defined as

$$\begin{array}{l} \mathsf{widen}(\sigma_1, \sigma_2) =_{df} \\ \quad \textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in} \\ \quad\quad \textbf{match } \sigma_1', \sigma_2' \textbf{ with } (\exists x_1^* \cdot \kappa_1 \wedge \pi_1), (\exists x_2^* \cdot \kappa_2 \wedge \pi_2) \textbf{ in} \\ \quad\quad\quad \textbf{if } \kappa_1 \vdash \kappa_2 * \texttt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{widen}_\pi(\pi_1, \pi_2)) \\ \quad\quad\quad \textbf{else } \sigma_1 \vee \sigma_2 \end{array}$$

where the rename function has the same effect as above. Generally this operator is analogous to join; the only difference is that we expect the second operand σ_2 is weaker than the first σ_1, such that the widening reflects the trend of such weakening from σ_1 to σ_2. In this case it applies the widening operation $\mathsf{widen}_\pi(\pi_1, \pi_2)$ over the numerical domain [23]. Therefore, based on the widening over conjunctive abstract states, we lift the operator over (disjunctive) abstract states:

$$\mathsf{widen}^\dagger(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\bigvee_i \sigma_i^1), (\bigvee_j \sigma_j^2) \textbf{ in} \bigvee_{i,j} \mathsf{widen}(\sigma_i^1, \sigma_j^2)$$

which is similar as its counterpart of the join operator. These three operations provides termination guarantee while preserving soundness, as the following example demonstrates.

Example 1 (Abstraction, join and widening). Assume we have two abstract states, $\Delta_0 = \mathtt{x}{::}\mathtt{node}\langle _, \mathtt{x_0}\rangle * \mathtt{x_0}{::}\mathtt{node}\langle _, \mathtt{null}\rangle$ and $\Delta_1 = \mathtt{x}{::}\mathtt{node}\langle _, \mathtt{x_0}\rangle * \mathtt{x_0}{::}\mathtt{node}\langle _, \mathtt{x_1}\rangle * \mathtt{x_1}{::}\mathtt{node}\langle _, \mathtt{null}\rangle$. We would like to discover a sound approximation for both states. Firstly we perform abstractions on both to obtain two abstract states, say, $\Delta_0' = \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_0}\rangle \wedge \mathtt{n_0}{=}2$ and $\Delta_1' = \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_0}\rangle \wedge \mathtt{n_0}{=}3$. Then these two are joined together according to shape similarity to be $\Delta_1'' = \mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_0}\rangle \wedge (\mathtt{n_0}{=}2 \vee \mathtt{n_0}{=}3)$, which transfers disjunction to numerical domain. Finally the joined state is widened based on the first state Δ_0', yielding a state $\mathtt{x}{::}\mathtt{ll}\langle \mathtt{n_0}\rangle \wedge \mathtt{n_0}{\geq}2$. It is a sound abstraction of both Δ_0 and Δ_1, and finishes the analysis with one more iteration.

Soundness and termination. The soundness of our analysis is ensured by the soundness of the following: the entailment prover [21], the abstract semantics (w.r.t. concrete semantics), the abstraction operation over shapes, and the join and widening operators.

Theorem 1 (Soundness). *Our analysis is sound due to soundness of entailment checking, abstract semantics, operations of abstraction, join and widening.*

The proof for entailment checking is by structural induction [21]; for abstract semantics is by induction over program constructors; for abstraction follows directly the first two; and for join and widening is based on entailment checking and soundness of corresponding numerical operators.

For the termination aspect, we have the result:

Theorem 2 (Termination). *The iteration of our fixpoint computation will terminate in finite steps for finite input of program and specification.*

The proof is based on two facts: the finiteness over the shape domain provided by our restriction on cutpoints, and the termination over the numerical domain guaranteed by our widening operator. The first can be proved by claiming the finiteness of all possible abstract states only with the shape information: recalling our analysis algorithm where we set an upper bound n for shared cutpoints (logical variables) we keep in track of, we know that the program and logical variables preserved in our analysis are finite. Meanwhile all possible shape predicates are limited; therefore all the shape-only abstract states are finite. The second is proved in the abstract interpretation frameworks for numerical domains [23]. These two facts guarantee the convergence of our analysis.

5 Experiments and Evaluation

We have implemented a prototype system for evaluation purpose. The prototype system was built in Objective Caml. We used SLEEK [21] as the solver for entailment checking over the heap domain, and Omega constraint solver [24] and Fixcalc solver [23] for join and widening operations in the numerical domain. Our test platform was an Intel Core 2 CPU 2.66GHz system with 8Gb RAM.

Program	Function	Time
`create`	Creates a list with given length parameter	0.452
`ins_sort`	Inner loop of Fig. 1	0.824
`ins_sort`	Outer loop of Fig. 1	4.372
`delete`	Disposes a list	0.720
`traverse`	Traverses a list	0.636
`append`	Appends two lists	0.312
`partition`	Auxiliary operation used by Quick-sort	1.497
`merge`	Merges two sorted lists to be one sorted list	1.972
`split`	Divides a list into two sublists with length difference of at most one	0.354
`select`	Selects the smallest node of a list	0.692
`select_sort`	Outer loop of selection sort	4.892
`tree_insert`	Inserts a node into a binary search tree	1.364
`tree_search`	Finds a node in a binary search tree	1.294

Fig. 5. Selected Experimental Results

Figure 5 describes the programs with which we performed experiments. The first column denotes the names of the programs. The second column states the programs' functionalities. The last column exhibits the time in second taken by our analysis. As can be seen from their functions, these programs involve recursive data structures such as (sorted) linked lists and binary (search) trees, and employ loops to manipulate these data structures (and some of them even have nested loops). Our target is to verify these programs with the help of our analysis over the loops they invoke, such that user annotations for while loops can be avoided. Our experiments have confirmed that HIP/SLEEK can verify all these programs successfully when supplied with loop invariants discovered by our analysis. According to our experience, these experiments just require the bound of shared cutpoints be a reasonably small number, say no more than twice of the number of program variables.

We have two main observations from our experimental results. The first is that we can handle many data structures with rich program properties they bear. To analyse these loops, we need to deal with both the list and list segment predicates to capture the linked list data structure, as well as their sorted version for the sorting algorithms. We can also handle tree-like predicates such as binary trees and binary search trees. Meanwhile these predicates also come along with many properties such as the length of the list and size/height of the tree, and the

minimum/maximum value of a sorted list/binary search tree. Based on them, our analysis is capable of expressing the invariants of these properties in terms of the constraints over the predicates' parameters.

Beyond the number of predicates and properties we can process, another observation on our analysis is that we can process them *rather precisely.* For example the list creation program creates a list with the same length as user input, and list traverse does not change list's length. Besides these, some loops provide critical invariants for the method running them to function correctly. For example, the quicksort algorithm partitions a list into three parts, where two are lists and the third just one node, whose value is exactly in the middle of that of the two other lists (`partition` in the table). We use a list bound predicate to indicate that fact which is successfully inferred by our analysis. We can also infer that the first loop of a mergesort (`split` in the table) can divide the list into two whose length difference is at most one, which is unimportant for the algorithm's functional correctness but essential for its performance. For `tree_insert`, we have the result that the tree's height is increased at most one, and the minimum/maximum value of the new binary search tree will be exactly the inserted value, if that value is out of the value bounds of the original tree. Such invariants are sufficiently precise to prove the functional correctness of all these programs with the given predicates.

6 Related Work and Conclusion

Related works. For heap-manipulating programs with any form of recursion (be it loop or recursive method call), dramatic advances have been made in synthesising their invariants/specifications. The local shape analysis [9] infers loop invariants for list-processing programs, followed by the SpaceInvader tool to infer full method specifications over the separation domain, so as to verify pointer safety for larger industrial codes [3,27]. The SLAyer tool [10] implements an inter-procedural analysis for programs with shape information. To deal with also size information (such as number of nodes in lists/trees), THOR [19] derives a numerical program from the original heap-processing one in a sound way, such that the size information can be obtained with a traditional loop invariant synthesis. A similar approach [11] combines a set domain (for shape) with its cardinality domain (for corresponding numerical information) in a more general framework. Compared with these works, our approach can handle data structures with stronger invariants such as sortedness and binary search property, which have not been addressed in the previous works.

One more work to be mentioned is the relational inductive shape analysis [4]. It employs inductive checkers to express both shape and numerical information. Our approach has four advantages over theirs: firstly, we try to keep as many as possible shared cutpoints (logical variables) during the analysis (within a preset bound), whereas they do not preserve such cutpoints (which is witnessed by their joining rules over the shape domain). Therefore our analysis is essentially more precise than theirs, e.g. in the second scenario of (6) described in Section 4.2.

Meanwhile, our approach can deal with data structures with loops in them (say cyclic linked-lists), whereas they do not have a mechanism to handle it. An example in point is the state $\mathtt{x::ls}\langle\mathtt{m},\mathtt{y}\rangle * \mathtt{y::ls}\langle\mathtt{y},\mathtt{n}\rangle \wedge \mathtt{n}>0$ involving both a shared cutpoint y and a circled list $\mathtt{y::ls}\langle\mathtt{y},\mathtt{n}\rangle \wedge \mathtt{n}>0$, neither of which can be handled by their work (while ours is capable of that). Another advantage of our approach over theirs is that they only demonstrate how to analyse a program with one particular shape, such as their examples analysing programs which manipulate binary search trees and red-black trees without changing the variety of shape in the heap. Comparatively, we allow different predicates to appear in the analysis of one program, like in our motivating example (thanks to our more flexible abstraction operation). Lastly, their work is mainly from a theory perspective as they do not employ numerical reasoners to solve the relational constraints in their implementation; on the contrary, we discharge all the numerical and relational constraints with automated reasoners [23,24].

There are also many other approaches that can synthesise shape-related program invariants than those based on separation logic. The shape analysis framework TVLA [26] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [12] reported a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. [16] developed a role system to express and track referencing relationships among objects, where an object's role (type) depends on, and changes according to, the mutation of its referencing. Hackett and Rugina [13] can deal with AVL-trees but is customised to handle only tree-like structures with height property. Compared with these works, separation logic based approach benefits from the frame rule and hence supports local reasoning.

Classical abstract interpretation [7] and its applications such as automated assertion discovery [8,15,17] mainly focus on finding numerical program properties. Compared with their works, ours is also founded on the abstract interpretation framework but tries to discover loop invariants with *both* separation and numerical information. Meanwhile, we can also utilise their techniques of join and widening to reason about the numerical domain, as we did for the work [23].

Concluding Remarks. We have reported an analysis which allows us to synthesise sound and useful loop invariants over a combined separation and numerical domain. The key components of our analysis include novel operations for abstraction, join and widening in the combined domain. We have built a prototype system and the initial experimental results are encouraging.

Acknowledgement. This work was supported by EPSRC Projects EP/G042322/1 and EP/E021948/1 and MoE Tier-2 Project R-252-000-444-112. We thank Florin Craciun for his precious comments.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2005)

2. Berdine, J., Cook, B., Distefano, D., O'Hearn, P.: Automatic termination proofs for programs with shape-shifting heaps. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 386–400. Springer, Heidelberg (2006)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: 36th POPL (January 2009)
4. Chang, B., Rival, X.: Relational inductive shape analysis. In: POPL (2008)
5. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties. In: 12th ICECCS (2007)
6. Chin, W.-N., David, C., Nguyen, H.H., Qin, S.: Enhancing modular oo verification with separation logic. In: POPL (January 2008)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
8. Cousot, P., Cousot, R.: On abstraction in software verification. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 37. Springer, Heidelberg (2002)
9. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
10. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
11. Gulwani, S., Lev-Ami, T., Sagiv, M.: A Combination Framework for Tracking Partition Sizes. In: POPL (2009)
12. Guo, B., Vachharajani, N., August, D.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
13. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL (2005)
14. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: POPL (2001)
15. Kovács, L., Jebelean, T.: An algorithm for automated generation of invariants for loops with conditionals. In: SYNASC (Symbolic and Numeric Algorithms for Scientific Computing) (2005)
16. Kuncak, V., Lam, P., Rinard, M.: Role analysis. In: POPL (2002)
17. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
18. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. To appear at LPAR-16 (2010)
19. Magill, S., Tsai, M., Lee, P., Tsay, Y.: Thor: A tool for reasoning about shape and arithmetic. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 428–432. Springer, Heidelberg (2008)
20. Nguyen, H.H., Chin, W.-N.: Enhancing program verification with lemmas. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 355–369. Springer, Heidelberg (2008)
21. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
22. Parkinson, M., Bierman, G.: Separation logic, abstraction and inheritance. In: POPL (2008)
23. Popeea, C., Chin, W.-N.: Inferring disjunctive postconditions. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 331–345. Springer, Heidelberg (2008)

24. Pugh, P.: The Omega Test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM (1992)
25. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: 17th LICS (2002)
26. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24(3), 217–298 (2002)
27. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)

Software Metrics in Static Program Analysis

Andreas Vogelsang[1], Ansgar Fehnker[2], Ralf Huuck[2], and Wolfgang Reif[3]

[1] Fakultät für Informatik, Technische Universität München
Boltzmannstr. 3, 85748 Garching b. München, Germany
andreas.vogelsang@in.tum.de

[2] National ICT Australia Ltd. (NICTA)* and University of New South Wales
Locked Bag 6016, Sydney NSW 1466, Australia
{ansgar.fehnker,ralf.huuck}@nicta.com.au

[3] Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg
Universtitätsstrasse 14, 86135 Augsburg, Germany
reif@informatik.uni-augsburg.de

Abstract. Software metrics play an important role in the management of professional software projects. Metrics are used, e.g., to track development progress, to measure restructuring impact and to estimate code quality. They are most beneficial if they can be computed continuously at development time. This work presents a framework and an implementation for integrating metric computations into static program analysis. The contributions are a language and formal semantics for user-definable metrics, an implementation and integration in the existing static analysis tool Goanna, and a user-definable visualization approach to display metrics results. Moreover, we report our experiences on a case study of a popular open source code base.

Keywords: software metrics, static program analysis, software quality, software maintenance.

1 Introduction

Many experts from academia as well as from industry would agree on the fact that most of today's software products and their development process are of comparatively low quality. The *2009 Standish Group CHAOS Report* [15] for example states that 24% of all software projects fail, which means they are cancelled prior to completion or delivered and never used, while only 32% can be considered as successful. One of the contributing factors is that modern software is almost never completely developed from scratch, but is rather extended and modified using existing code and often includes third party source code. This can lead to poor overall maintainability, difficult extensibility and high complexity. To better understand the impact of code changes and track complexity issues as well as code quality software metrics are frequently used in the software development life cycle.

* Funded through the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 485–500, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Ideally, software metrics should be computed continuously during the development process to enable the best possible tracking. Moreover, software metrics should be definable by development teams to not only cover general factors, but to measure company, project or team specific goals. In this work we present an integrated and flexible approach to metric computation by embedding it into static program analysis. As such, metrics can be computed on demand for every compilation even long before the software is fully developed.

In particular, we present a novel metric specification language (GMSL) that enables software developers to quickly specify their own metrics. We further define the formal syntax and semantics for GMSL, and implemented an interpreter that embeds the metric calculation in our existing static analyzer Goanna. On top of this we present a generic and user-definable visualization approach that enables quick tracking of metric results. Moreover, we report on our experiences integrating a metric specification language into static program analysis as well as our experiences from real world case studies.

Related to our approach are a number of tools that enable to compute metrics or query code for programming constructs. ODASA[1] is a commercial software assets analyzer that adds all software artefact's into a repository and provides a query engine to search for bottlenecks or quality flaws. Coverity Architecture Analysis[2] is a commercial static program analyzer for C/C++ and Java programs. It offers an architecture analysis and comes with predefined metrics that focus on complexity. Klocwork Insight[3] is another commercial source code analysis suite that includes an *Integration Build Reporting and Metrics* module for a large number of predefined metrics. NDepend[4] is a Visual Studio tool that helps the user to manage complex .NET code bases. NDepend considers the code as a database and the user can query the database and display the query results. SonarJ[5] is another software architecture management tool based on static analysis. Its main focus is to assure the consistency of the logical architecture of a system and its actual implementation. Additionally, SonarJ computes metrics, such as Robert Martin's metrics [10], and provides a histogram chart to visualize the development over time.

All of the mentioned tools can be partitioned into two different categories: Either offering a query language that allows the user to query his code for particular constructs or computing metric values on the source code during the build process based on pre-defined settings. None of the tools provide a mechanism that allows the user to define his or her own metrics that are subsequently computed automatically by the analysis tool in each compilation or build. Also, the visualizations are usually specific to the predefined metrics and measures. In contrast, our approach enables to link user-defined metrics to generic visualizations, which are independent from the metric's semantics.

[1] `http://semmle.com/technology/how-it-works/`

[2] `http://www.coverity.com/products/architecture-analysis.html`

[3] `http://www.klocwork.com/products/insight/`

[4] `http://www.ndepend.com/Metrics.aspx`

[5] `http://www.hello2morrow.com/products/sonarj/`

The next section introduces software quality metrics and static analysis, especially GOANNA. Section 3, and 4 cover the metric specification language GMSL, metric computation in GOANNA, and metric visualization. Section 5 discusses application of the tool to the AUDACITY code base, and its performance, while Section 6 concludes with an outlook on future work.

2 Integrating Software Metrics

Software metrics. Software metrics measure properties of software and are loosely defined in the IEEE 1061 standard [9] as

> "A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality. "

This means that metrics make a statement about some quality attributes, are quantitative, but will have to be interpreted by a human. In this work we focus on so called *software product metrics*, which covers the aspects of *size*, *complexity*, and *quality* that can be measured on the source code and its evolution over time. Example product metrics are *lines of code*, *cohesion*, *coupling* or *cyclomatic complexity*. We will go into detail in Section 3.

While there has been a substantial body of work on metrics definitions and their correlation with program faults [7,13,14] or maintainability and bugs [4,5] we will not discuss which metrics are reasonable or particularly important. Neither will we address which metric values indicate good or poor quality. Instead we are proposing a framework that allows to define all these metrics in a flexible and concise manner and integrate them into the standard compilation and source code analysis process.

Level of Abstraction. Metrics can be defined on various levels of abstraction. Common metrics such as McCabe's cyclomatic complexity [11] are defined on the *control flow graph (CFG)* of a program and can be stated as

$$CC = e - n + 2p, \quad (1)$$

where e is the number of edges, n is the number of nodes and p is the number of strongly connected components in the CFG. Implementations, however, are typically more language specific. The tool NDEPEND for example defines cyclomatic complexity as:

$$CC = 1 + \{\text{number of the following expressions found in a method}\} :$$
$$\texttt{if}|\texttt{while}|\texttt{for}|\texttt{foreach}|\texttt{case}|\texttt{default}|\texttt{continue}|\texttt{goto}|\texttt{\&\&}|\texttt{||}|\texttt{catch}|\texttt{?:}|\texttt{??} \quad (2)$$

This definition enumerates the concrete code constructs that contribute to cyclomatic complexity. These differ from language to language and the above definition is only valid for the programming language C#.

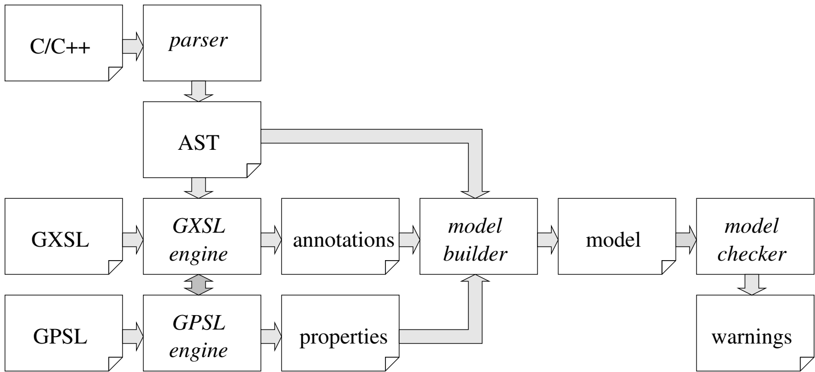


Fig. 1. GOANNA's model checking approach for statically analyzing C/C++ code

This work introduces an approach to define metrics on a more abstract level such as in (1). This means, the definition is closely related to its mathematical representation. This improves readability and maintainability of the metric definition itself. However, we also provide means to associate these definitions with elements in the *abstract syntax tree (AST)*, such that the metric definitions can be automatically computed for real-life source code.

Integrating Metric Computation. Metrics can be computed on their own or integrated into the compiler or existing source code analysis frameworks. Integration into existing frameworks leverages existing technology and requires fewer process changes for software development teams. This means, metric results are an added feature of tools that are already in frequent use.

In this work we integrate user-definable metrics in our static source code analyzer GOANNA. This tool performs deep analysis of C/C++ source code using model-checking [3] technology. GOANNA checks for bugs, memory leaks and security vulnerabilities, is fully path-sensitive and inter-procedural, and makes use of additional techniques such as abstract interpretation. A more detailed overview can be found in [6].

GOANNA already provides two specification languages for defining source code checks. The first language is a tree-query language based on XPath [2] for finding constructs and patterns of interest in the AST and is called *Goanna XPath Specification Language (GXSL).* The second language is based on temporal logic expressions over paths in the CFG and is called *Goanna Property Specification Language (GPSL).* GPSL allows the embedding of GXSL expression. An example is to query for `malloc` and `free` constructs in GXSL and then use the information to define in GPSL that all paths in the program from a `malloc` should lead to a `free`. Figure 1 shows how these languages feed into the static analysis. More details can be found in [16].

This work uses the existing framework and introduces a metric specification language that can reference to earlier query results, count, and compute metrics based on arithmetic expressions. The new language will be introduced in the next section.

3 Metric Specification Language GMSL

The *Goanna Metric Specification Language (GMSL)* provides a way to define metrics on an abstract level. A prerequisite for the use of GMSL is a query engine that returns sets of nodes of the AST for which certain syntactic properties hold. As mentioned in Section 2, GOANNA provides a language GXSL language to define functions that select certain nodes of the AST of a program. The queries are always evaluated on the entire AST but it is possible to pass parameters to the queries to refer to particular node (or sub-trees) in the AST. The result of a GXSL query is a set of AST nodes.

Most metrics are defined for a given scope, this means for a particular set of nodes in the AST. For example, a metric might be defined for the scope *all_classes*, which means that one metric value will be computed for each class. And each class in the programm corresponds to a sub-tree in the AST. Other metrics are defined for scopes like *functions* or *namespaces*. In GMSL the scope of a metric is mention in its definition, and metric values will be computed for every instance of the scope.

GMSL distinguishes between two types of variables. One ranges over nodes (or sub-trees) of the AST, and the values are obtained by GXSL queries on the AST or sub-trees of it. These variables will be passed as arguments to other GSXL queries. The other type of variable represents integer and real numbers, which either represent the cardinality of sets, results obtained from other metrics, the result of an arithmetic expression, or the aggregated result of those. For simplicity we assume that these numbers are reals. The actual definition of the metric then is a mathematical expression containing variables over the reals, queries and constants.

3.1 Syntax

The grammar of GMSL, given in *Extended Backus Naur Form (EBNF)*, is defined in Table 1. Before we introduce the semantics, we first provide a few examples for common metrics to illustrate the language. A few functions are used in these examples, which are provided by GOANNA's AST query library. This library can be extended by user-defined AST queries, e.g. GXSL functions, defined specifically to compute metrics. The following examples also demonstrate how to define a wide variety of metrics found in literature.

Cyclomatic Complexity. Cyclomatic Complexity of a function as defined in NDEPEND is the number of branches in the control flow of a function plus one. If we only consider one function, i.e. one strongly connected component of the corresponding CFG, this definition is equal to McCabe's definition [11], which

Table 1. GMSL Grammar in EBNF

```
gmsl          = "METRIC" name scope [venv] definition ;
scope         = '('  node  "IN" function ')' ;
name          = ident ;
venv          = "WITH" vdecl (',' vdecl)* ;
vdecl         = var '=' binding ;
definition    = "DEF" expression ;
binding       = function | aggregator function "OVER" setindex ;
aggregator    = "SUM" | "MAX" | "MIN" | "PROD" ;
setindex      = node  "IN"  function ;
function      = ident  [ '('  [ ident (',' ident)* ] ')' ] ;
expression    = var | function | num | expression op expression ;
op            = '+' | '-' | '*' | '/' ;
var           = '@' ident ;
node          = ident ;
num           = nat | real ;
nat           = ( '0' | ... | '9' )+ ;
real          = nat '.' nat ;
ident         = ( 'a' | 'b' | ... | 'Z' | '_')+ ;
```

defines the cyclomatic complexity as the number of linearly independent paths in the control flow of a function:

```
METRIC cc_per_f (f IN all_funs)
WITH @cn = all_cond_nodes(f)
DEF 1 + @cn
```

The metric will be computed for all nodes f returned by the GXSL query *all_funs*. It is defined as:

```
fun all_funs()
    <<./FunDecl>>
```

This function returns the corresponding AST node for every function of a given program. The metric value of f is determined by the number of conditional nodes in f, given by the GXSL query *all_cond_nodes*, plus one. The query *all_cond_nodes* lists all conditional nodes, similar to definition (2), for C/C++:

```
fun all_cond_nodes(f)
  f<< .//If | .//While | .//For | .//Goto | .//Label | .//Default |
      .//Op2[@op='LogicalOr' or @op='LogicalAnd'] | .//Handler |
      .//Op3[@op='Cond']>>
```

Afferent Coupling. Afferent Coupling of a class as defined by ARiSA[6] is the number of classes that call a certain class:

[6] http://www.arisa.se/compendium/node104.html

```
METRIC afferent_coupling (c IN all_classes)
WITH @ca = SUM dependency(g,c) OVER g IN all_classes
DEF @ca
```

The metric will be computed for all nodes c returned by the AST query $all_classes$. The metric value of c is determined by the sum of $dependency(g, c)$, applied to all nodes g, returned by the AST query $all_classes$. The AST query $dependency(g, c)$ returns one node for class g, if there is a function call in class g to class c.

Cohesion. Cohesion of a class as defined in [1] is a measure of how strongly-related and focused the various tasks of a class are, depending on how many methods of a class access common fields or call common other methods of the same class:

```
METRIC cohesion (c IN all_classes)
WITH @N = methods_of_class(c),
     @E = SUM directly_related(m) OVER m IN methods_of_class(c)
DEF @E /(@N * (@N-1))
```

The metric will be computed for all nodes c returned by the AST query $all_classes$. The AST query $directly_related(m)$ returns a node for all methods of the same class that are directly related to method m (i.e. they both access a certain common field or they are both calling another common method of the class). If every method is directly related to all other methods, then the metric value is equal to 1.

3.2 Semantics

The semantics of GMSL will be given as a denotational semantics which uses *environments* to map syntax to semantics. There are four types of environments:

- $\varsigma \in GXSLLib$ is a GXSL environment which maps GXSL function names to the actual GXSL functions.
- $\mu \in MEnv$ is a metric environment that maps metric names to their semantic function.
- $\eta \in NEnv$ is a node environment which maps node variables to their corresponding AST node.
- $\nu \in VENV$ is a variable environment which maps counting variables to their semantic value.

These environments and their product, which is denoted by Env are used to define the semantics of GMSL.

The semantics are defined via a function $\mathcal{M}$, which compiles a metric definition to a metric environment. All information that are necessary for applying a metric definition to a program are contained in that metric environment.

$$\mathcal{M}[\![-]\!] : MDecl \rightarrow GXSLLib \times MEnv \rightarrow MEnv \tag{3}$$

$$\mathcal{M}[\![m]\!](\varsigma, \mu) = \mu\,[name(m) \mapsto \mathcal{S}[\![m]\!](\varsigma, \mu, \emptyset, \emptyset)] \tag{4}$$

Function $\mathcal{S}$ maps, given an initial environment, the environment to a function that takes a program and maps the nodes of this program that are within the scope of the metric to real numbers. It is defined as follows.

$$\mathcal{S}[\![-]\!] : \mathit{MDecl} \rightarrow (\mathit{Env} \rightarrow (\Pi p : \mathit{Prog}\,.\,\mathit{nodes}(p) \rightharpoonup \mathbb{R})) \quad (5)$$

$$\mathcal{S}[\![\text{METRIC } \mathit{name}\ (\mathit{scope} \text{ IN } f)\ \mathit{venv}\ \mathit{definition}]\!](\varsigma, \mu, \eta, \nu) = \quad (6)$$

$$\lambda p \in \mathit{Prog}\,.\,\lambda n \in \mathcal{G}[\![f]\!](\varsigma, \eta)(p)\,. \quad (7)$$

$$\mathcal{D}[\![\mathit{definition}]\!]\,(\mathit{upd}_V(\mathit{venv})(\varsigma, \mu, \eta[\mathit{scope} \mapsto n], \nu)(p))\,(p) \quad (8)$$

This definition reflects that a metric encompasses a *scope*, a *declaration* of counting variables, and an *arithmetic expression* over variables and applications of GMSL and GXSL functions. The set $\mathcal{G}[\![f]\!](\varsigma, \eta)(p)$ in (7) contains all scope instances. Function $\mathcal{G}$ is defined by the GXSL semantics, and returns for a given environment a set of AST nodes. Given the variable declaration part, upd_V in (8) updates $\nu \in \mathit{VEnv}$ such that it maps the counting variables to the semantics $\mathcal{B}$ of the associated *binding*. Function $\mathcal{D}$ associates the metric with the semantics $\mathcal{E}$ for the associated arithmetic expression. We omit the formal definition of $\mathcal{D}$, and upd_V for brevity; $\mathcal{E}$ will be defined below. The semantics of the bindings are defined as follows:

$$\mathcal{B}[\![-]\!] : \mathit{binding} \rightarrow (\mathit{GXSLLib} \times \mathit{MEnv} \times \mathit{NEnv} \rightarrow (\mathit{Prog} \rightarrow \mathbb{R})) \quad (9)$$

$$\mathcal{B}[\![f]\!](\varsigma, \mu, \eta) = \lambda p \in \mathit{Prog}\,.\,\mathcal{F}[\![f]\!](\varsigma, \mu, \eta)(p) \quad (10)$$

$$\mathcal{B}[\![\text{SUM } f \text{ OVER } \mathit{node} \text{ IN } g]\!](\varsigma, \mu, \eta) = \quad (11)$$

$$\lambda p \in \mathit{Prog}\,.\sum_{n \in \mathcal{G}[\![g]\!](\varsigma, \eta)(p)} \mathcal{F}[\![f]\!](\varsigma, \mu, \eta[\mathit{node} \mapsto n])(p) \quad (12)$$

The semantics of the remaining aggregators *PROD, MAX, MIN* are defined analogously. A binding of a counting variable can either be a simple function or an aggregation over a set of numbers determined by the application of a function on the results of a node set, returned by another function. Simple functions in this case can be GXSL query functions from the library or the name of another GMSL metric. The semantics of a simple function f is determined by the semantic function $\mathcal{F}$. If f is a GXSL library function, $\mathcal{F}[\![f]\!](\varsigma, \mu, \eta)(p)$ in (10) or (12) returns the cardinality of the associated set. If f is a GMSL library function, it returns a real number representing a metric value.

$$\mathcal{F}[\![-]\!] : \mathit{function} \rightarrow (\mathit{GXSLLib} \times \mathit{MEnv} \times \mathit{NEnv} \rightarrow (\mathit{Prog} \rightarrow \mathbb{R}))$$
$$\mathcal{F}[\![\mathit{libfun}(n_1, \ldots, n_k)]\!](\varsigma, \mu, \eta) = \lambda p \in \mathit{Prog}\,.\,|\mathcal{G}[\![\mathit{libfun}(n_1, \ldots, n_k)]\!](\varsigma, \eta)(p)|$$
$$\mathcal{F}[\![\mathit{metric}(n)]\!](\varsigma, \mu, \eta) = \lambda p \in \mathit{Prog}\,.\,\mu(\mathit{metric})(p)(\eta(n)(p))$$

The arithmetic expression is the *definition* in semantic function $\mathcal{S}$. The semantics of these arithmetic expressions are defined as follows:

$$\mathcal{E}[\![-]\!] : \textit{definition} \rightarrow (\textit{Env} \rightarrow (\textit{Prog} \rightarrow \mathbb{R}))$$
$$\mathcal{E}[\![@v]\!](\varsigma, \mu, \eta, \nu) = \lambda p \in \textit{Prog} \,.\, \nu(@v)(p)$$
$$\mathcal{E}[\![n]\!](\varsigma, \mu, \eta, \nu) = \lambda p \in \textit{Prog} \,.\, \mathcal{N}(n)$$
$$\mathcal{E}[\![\textit{exp}_1 + \textit{exp}_2]\!](\varsigma, \mu, \eta, \nu) =$$
$$\lambda p \in \textit{Prog} \,.\, \mathcal{E}[\![\textit{exp}_1]\!](\varsigma, \mu, \eta, \nu)(p) + \mathcal{E}[\![\textit{exp}_2]\!](\varsigma, \mu, \eta, \nu)(p)$$

The semantics of the remaining mathematical operators $-, *, /$ are defined analogously. An expression in a definition can either be a counting variable, a constant number or a composition of expressions. If the expression is a counting variable, the semantics of it is just the semantics of the *binding* to which it is mapped in the counting variable environment.

Example. To illustrate the defined semantics consider the following metric definitions:

```
METRIC avg_method_cc (c IN all_classes)
WITH @s = SUM cc_per_f(m) OVER m IN methods_of_class(c),
     @n = methods_of_class(c)
DEF @s / @n
```

This metric `avg_method_cc` computes the average cyclomatic complexity of the methods of a class. The functions `all_classes` and `methods_of_class(c)` return the set of all class nodes (sub-tree), or for a given class node (sub-tree) the set of all method nodes (sub-trees). Function `cc_per_f(m)` is a call to another GMSL metric that computes the cyclomatic complexity per function. This metric was defined on page 489. We apply this metric definition to the following C++ program:

```
class Number{
  private:    int n;
  public:     Number(int number){n=number;}
              void inc();
              void dec();};

void Number::inc(){ n++;}

void Number::dec(){ if (n>0) n--;}

int main(){  return 0;}
```

This C++ program consists of one class with two public methods and one constructor and a *main* function. Since *Number* :: *dec*() has a branching condition its cyclomatic complexity is 2; the cyclomatic complexity of all other functions is 1. Class `Number` is in the set returned by the GXSL query `all_classes` (applied to the program), thus within its scope.

Variable `@s` has value $\sum_{\mathtt{m} \in \mathcal{G}[\mathtt{methods_of_class(c)}]} \mathcal{M}[\mathtt{cc_per_f}](\mathtt{m})$, i.e 4. Variable `@n` has value $|\mathcal{G}[\mathtt{methods_of_class(c)}]|$, i.e 3 as there are three methods. Hence, the expression `@s/@n` evaluates to an average cyclomatic complexity of $1\,\frac{1}{3}$.

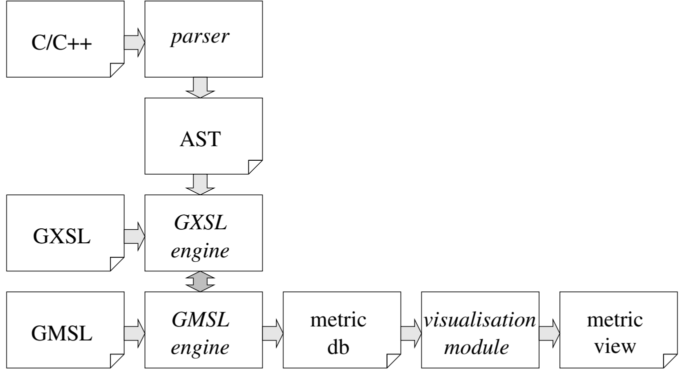


Fig. 2. Goanna's architecture for metric computation

4 Metric Module

4.1 GMSL Interpreter

The Goanna GMSL interpreter is an extension to the existing Goanna analyzer. An overview of the extended architecture can be found in Figure 2. The metrics interpreter sits on top of the existing GXSL query engine, i.e., mostly uses existing library functions for pattern matching constructs of interest, and interprets the metric specification written in GMSL. Metric specifications are written in text files and that way passed to the metric module.

From an implementation point of view it is interesting to note that some metrics are incrementally computed during an analysis run with the help of a database. The reason is as follows: Some metrics require more information than what can be gathered from a local function or a single file. For instance, to compute the number of method instances of a class or computing the number of calling functions for a given callee typically requires to aggregate information from the whole project. Therefore, we use a database to store partial information where necessary and aggregate this information during the analysis of the whole program.

4.2 Visualization Module

The previous sections covered the definition and computation of metrics. However, as mentioned in Section 2 software metrics are meant to be interpreted by humans. To assist the judging process and help to understand the data we define a generic visualization model. This enables a number of different views for a given set of metric values and allows the visualization of any user-defined metric.

To assist interpretations of the data, users of Goanna[]'s metric module can specify information which will be used in tooltip, comments, and most importantly, to properly scale the different metrics. For the latter we implemented

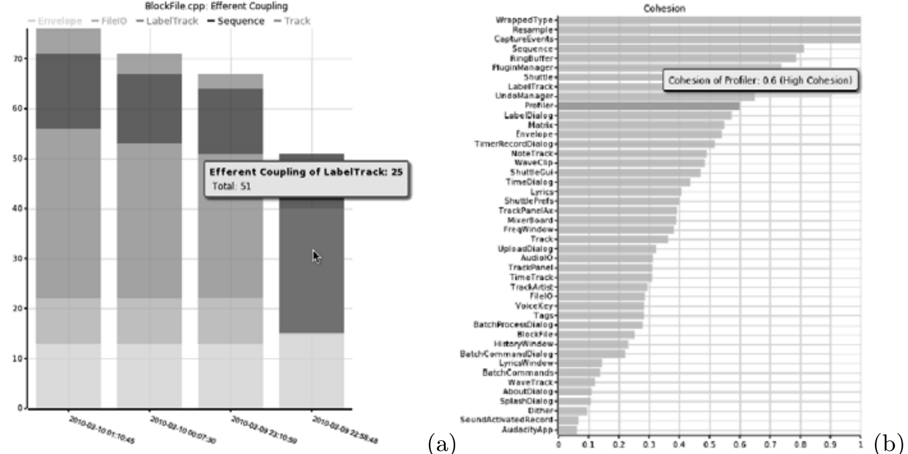


Fig. 3. (a) Histogram implementation of the time view. The histogram shows the efferent coupling over time for different classes. (b)Bar chart implementation of the metric view. Ranking of classes by cohesion.

a user-defined mapping of GMSL output to a finite number of categories. For instance, the following ranges and categories were defined for cyclomatic complexity:

```
= 1   : No Branching
1-15  : Easy
15-30 : Hard to Maintain
> 30  : Extremely Complex
```

These categories can be used as the visualization domain for different views, and aid the interpretation of the results.

In the following we describe the four views for metric visualization implemented in Goanna. We say $S = (M, t)$, is a *snapshot* of a project, where M is a set of GMSL metrics and t is a time stamp.

Time view: The time view is a sequence of program snapshots ordered by their time stamps. Given a sequence of snapshots $(M_0, t_0), \ldots, (M_n, t_n)$ the time view will display for each time stamp all chosen metric results per scope in M_i. This provides a good overview of how different metric values change over time. In the visualization module this will be displayed as a stacked bar chart as seen in Figure 3(a).

Metric view: The metric view is the summary of one metric for all elements in one scopes at one point in time, i.e., for a single snapshot (M, t). In Goanna the metric view is implemented by a horizontal bar chart that lists the metric values of different elements in the scope in decreasing order. Figure 3(b) shows an example for the ranking of classes by cohesion.

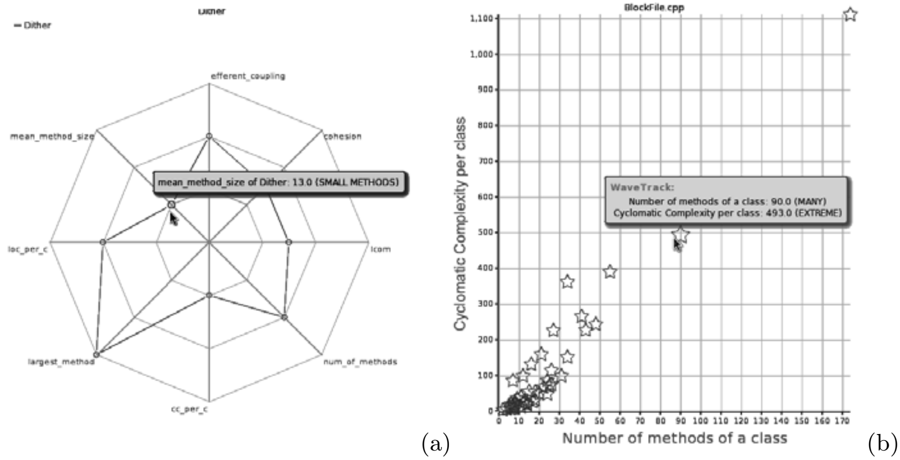


Fig. 4. (a) Radar chart implementation of the scope view. All metric values for a given class. (b) X-Y-Plot for the correlation view. This figure correlates the number of methods of a class, with the cyclomatic complexity.

Scope view: The scope view is the summary of all metric values that are computed for a certain instance of a scope at a certain time. The scope view is implemented by a radar chart where every axis represents a metric. An example for the different metric values of a given class is given in Figure 4(a).

Correlation view: The correlation view is a combination of the metric view and the scope view. It enables the user to examine how the values of a pair of metrics correlate over several scope instances. The correlation view is implemented in the form of an X-Y-Plot. See Figure 4(b) for an example.

The different metric views are configurable and can be combined in a dashboard if desired, but most importantly they are independent from a metric itself. As such they can visualize any metric and sufficiently provide a quick overview of the status of a software project.

5 Case Study

This section reports on the application of Goanna's metric module to the *Audacity*[7] code base. Audacity is an open source audio editor and written in C++. The latter was essential for testing the metrics defined for classes. With about 90,000 lines of code it has a reasonable size, and is, with around 70 million total downloads on *sourceforge.net*, also quite popular. The tests were performed on a desktop PC with 4 GB RAM and an Intel Core 2 Quad CPU @ 2.66 Mhz. The results for an implementation of the metric module based on Goanna version 1.1.

[7] `http://audacity.sourceforge.net/`

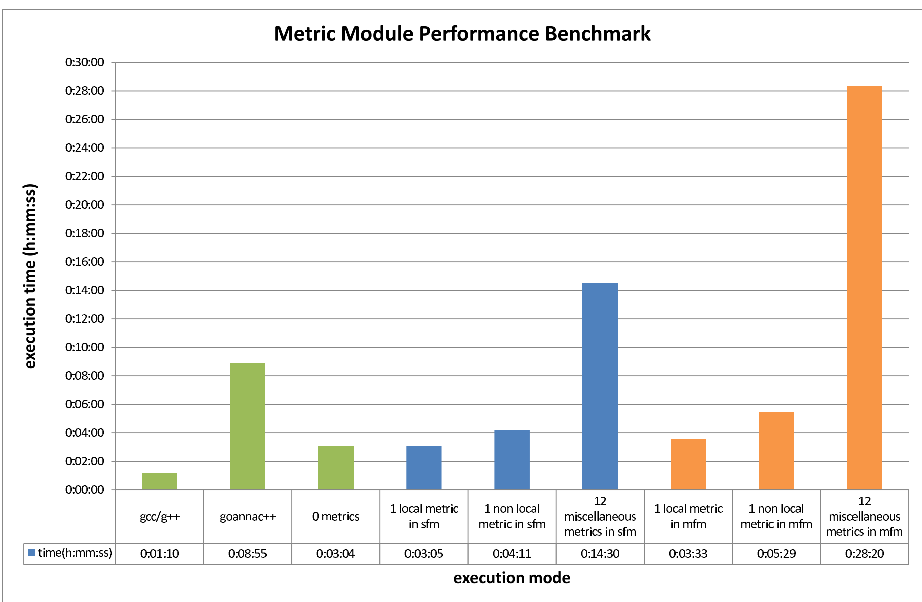


Fig. 5. Runtimes of GOANNA version 1.1 in different modes on the Audacity code base

The original build process of Audacity uses GCC to compile and link the source code. This build process takes 1:10 minutes to complete. The runtime of the metric module will be composed of: this compile time (because GOANNA also compiles the code), the time to extract the AST of the source files, the parsing of the metric definitions, and the metric computation itself. To separate the computation from the parsing steps, the module was run with an empty metric definition. Compiling the source code and extracting the AST took 03:04 minutes.

To measure and profile the performance of the metric computation, we set up six different test cases. These test runs are combinations of using one local metric, one non-local metric, and twelve miscellaneous metrics. Moreover, each of these cases were run in single file mode (sfm) and multiple file mode (mfm). A local metric is a metric that uses only queries that can be evaluated directly on the local scope instance. For instance, the metric *number_of_methods* is a local metric. A non-local metric, in contrast, iterates over sets of nodes that span multiple files. Metric *avg_method_cc* is an example, since it iterates over the set of methods of a class, which may be distributed over multiple files.

Among the twelve metric we measured were: *Cyclomatic complexity*, *Afferent coupling*, *Efferent coupling*, and *Instability* [8] of classes and functions, and *Lack of cohesion in methods of a class (LCOM)*.

The runtimes of these tests as well as the above mentioned runtimes for Gcc and the Goanna's bug detection (goannac++) are shown in Figure 5.

One immediate observation is that the runtimes heavily depend on the number, kind, and complexity of the GXSL functions used. As shown by the difference in runtime between the computation of a local metric and a non local metric, the use of aggregations takes significantly longer. This is due to the iteration over node sets, which may result in quadratic runtime, instead of linear in terms of node instances. On the other hand the evaluation of GXSL queries, especially on large ASTs, took the biggest proportion of time.

Another observation is that when running Goanna in multiple file mode (mfm) for one metric the runtime only increased by around 15-30% in comparison to the single file mode (sfm), the runtime for 12 metrics roughly doubled. This overhead can be explained by three reasons: Firstly, in multiple file mode all query results are stored in a database. Hence, every application of a query causes some additional database operations. Secondly, an aggregation in multiple file mode can be more expensive, because the aggregation set is typically larger. The third reason for the overhead had to do with slow string operations that were used for the communication with the database.

Some of the performance issues have been addressed in later versions of Goanna, but we like to point out that the current implementation is a prototype and has a lot of room for improvement. What is more important is that we were able to easily specify metrics and experimentally confirm some of the arguments brought forward in the literature as we see next.

Notable Results. The results we obtained were compared to some claims made by other authors. For instance, McConnell [12] classifies modules that handle all I/O routines as *logical cohesive*. In his system of seven cohesion classes logical cohesion is the second worst. Audacity has two I/O classes, named *AudioIO* and *FileIO*. The results obtained by the metric module confirm McConnell's conjecture: The cohesion computed by Goanna according to Badri's [1] formula resulted in 0.28 for *FileIO* and 0.3 for *AudioIO*, which is on the low end of the spectrum. The highest value of cohesion of the entire project had a class called *WrappedType*, which can be identified as functional cohesive. According to McConnell's classification, functional cohesion is the best category.

The correlation view of some values also revealed some expected connection between the metrics. As Figure 4(b) showed, there is a linear correlation between cyclomatic complexity of a class with an increasing number of methods in the Audacity code base. Of course, one simple contributing factor is that the addition of a method to a class will increase its cyclomatic complexity by at least one. Another observation is the correlation between *cohesion* and *LCOM*, which indicates the lack of cohesion of methods. As one might expect, an increasing cohesion value results in a decreasing lack of cohesion. The correlation view of these values for the Audacity code base is shown in Figure 6.

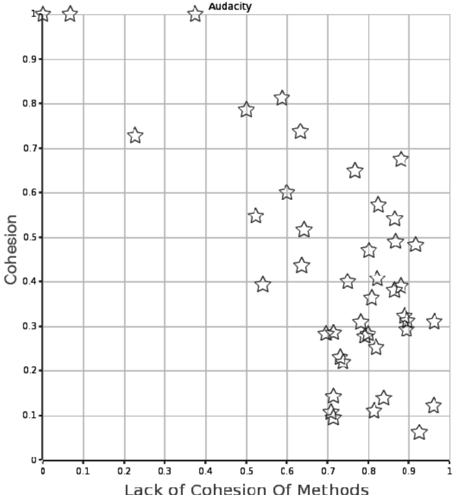


Fig. 6. Correlation of metric values of *cohesion* and *LCOM* (lack of cohesion of methods) on the Audacity code base

6 Conclusions

In this work we presented an approach to user-defined software metrics and a seamless integration into static program analysis. Unlike existing approaches the metrics are not hard coded, but interpreted at analysis time from a textual description that can be defined by software developers and teams themselves. The specification language GMSL is based on a formal syntax and semantics. While we chose to integrate the interpreter in our own tool there is in principle no restriction for using the same approach in, e.g., the standard compiler.

On top of the metric specification language we built the proof of concept of a generic metric visualization module. This module enables the mapping of any metric to different views and the automatic user-defined mapping of values to abstract categories. In practice, this has been proven useful to quickly assess the state of a software project.

Future work has to address some of the current implementation issues, such as relatively slow database access and optimizing the query interpretation. Moreover, some work has to go into scaling the used visualization techniques to large software projects. Once the user is confronted with dozens of metrics and thousands of files it is important to have some automated visual abstraction to avoid confusion and overload.

References

1. Badri, L., Badri, M.: A proposal of a new class cohesion criterion: An empirical study. Journal of Object Technology 3(4), 145–159 (2004)
2. Clark, J., DeRose, S.: XML Path Language 1.0 (XPath). W3C (1999), `http://www.w3.org/TR/xpath`

3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
4. Curtis, B., Sheppard, S.B., Milliman, P.: Third time charm: Stronger prediction of programmer performance by software complexity metrics. In: Proceedings of the Fourth International Conference on Software Engineering, pp. 356–360. IEEE Computer Society Press, Los Alamitos (1979)
5. Elshoff, J.: An analysis of some commercial PL/I programs. IEEE Transactions on Software Engineering SE-5(2), 113–120 (1976)
6. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model Checking Software at Compile Time. In: Proceedings of the 1st International Symposium on Theoretical Aspects of Software Engineering, Shanghai, China (2007)
7. Ferzund, J., Ahsan, S.N., Wotawa, F.: Empirical evaluation of hunk metrics as bug predictors. In: Abran, A., Braungarten, R., Dumke, R.R., Cuadrado-Gallego, J.J., Brunekreef, J. (eds.) IWSM 2009. LNCS, vol. 5891, pp. 242–254. Springer, Heidelberg (2009)
8. IBM: In pursuit of code quality: Code quality for software architects, Website `http://www.ibm.com/developerworks/java/library/j-cq04256/` (visited on February 3, 2010)
9. IEEE: IEEE Standard for a Software Quality Metrics Methodology. Institute of Electrical and Electronics Engineers (1061)
10. Martin, R.C.: Agile software development: principles, patterns, and practices. Alan Apt series. Prentice-Hall, Englewood Cliffs (2003)
11. McCabe, T.J.: A complexity measure. IEEE Transactions on Software Engineering 2(4), 308–320 (1976)
12. McConnell, S.: Code Complete: A Practical Handbook of Software Construction. Microsoft Press, Redmond (1993)
13. Misra, S.C., Bhavsar, V.C.: Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In: Kumar, V., Gavrilova, M.L., Tan, C.J.K., L'Ecuyer, P. (eds.) ICCSA 2003. LNCS, vol. 2667, pp. 724–732. Springer, Heidelberg (2003)
14. Nagappan, N., Ball, T., Zeller, A.: Mining metrics to predict component failures. In: ICSE 2006: Proceedings of the 28th International Conference on Software Engineering, pp. 452–461. ACM, New York (2006)
15. The Standish Group: Chaos report (2009), Website `http://www1.standishgroup.com/newsroom/chaos_2009.php` (visited on February 25, 2010)
16. Vistein, M., Ortmeier, F., Reif, W., Huuck, R., Fehnker, A.: An abstract specification language for static program analysis. Electr. Notes Theor. Comput. Sci. 254, 181–197 (2009)

A Combination of Forward and Backward Reachability Analysis Methods

Kazuhiro Ogata and Kokichi Futatsugi

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

Abstract. Induction-guided falsification (IGF) is a combination of bounded model checking (BMC) and structural induction, which can be used for falsification of invariants. IGF can also be regarded as a combination of forward and backward reachability analysis methods. This is because BMC is a forward reachability analysis method and structural induction can be regarded as a backward reachability analysis method. We report on a case study in which a variant of IGF has been used to systematically find a counterexample showing that NSPK does not enjoy the agreement property.

Keywords: agreement property, CafeOBJ, bounded model checking, falsification, NSPK, structural induction, Maude.

1 Introduction

Bounded model checking (BMC)[1] has been used to discover a counterexample showing that a hardware or software system does not enjoy a safety property. It (or its concept) has been adopted by some software analysis tools such as Alloy[2]. Basically it starts with some initial states of a system and exhaustively traverses the state space reachable from the initial states up to some specific depth. Therefore, BMC is a forward reachability analysis method.

A backward reachability analysis method starts with some states of a system such that a safety property is broken and traverses the state space reachable in a backward sense from the states. If it reaches an initial state of the system, the system does not enjoy the safety property. If the entire state space reachable in a backward sense from such an arbitrary state does not contain any initial states, the system enjoys the safety property. Among tools adopting a backward reachability analysis method is Maude-NPA[3].

We have proposed a combination of BMC and structural induction called induction-guided falsification (IGF)[4]. IGF uses as BMC the search functionality provided by CafeOBJ[5] and Maude[6]. In IGF, structural induction is conducted by human users by writing what are called proof scores in CafeOBJ. IGF can also be regarded as a combination of forward and backward reachability analysis methods. This is because structural induction can also be regarded as a backward reachability analysis method.

In this paper, we review IGF and report on a case study in which a variant of IGF, where non-necessary lemmas may be used, has been used to systematically

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 501–517, 2010.
© Springer-Verlag Berlin Heidelberg 2010

discover a counterexample showing that NSPK[7] does not enjoy the agreement property. Many case studies have been reported, systematically discovering a counterexample showing that NSPK does not enjoy the nonce secrecy property[8,9,3]. To our best knowledge, however, few case studies have been reported for the agreement property.

The rest of the paper is organized as follows. §2 describes the OTS/CafeOBJ method[10] for specification and verification of systems. A simple example is used to describe the method and demonstrate that (structural) induction can be used to falsify that a system enjoys a property. §3 describes the search functionality. §4 describes a viewpoint that regards (structural) induction as a backward reachability analysis method, reviews IGF, and introduces a variant of IGF. §5 reports on the case study. §6 mentions some related work. §7 concludes the paper.

2 The OTS/CafeOBJ Method

2.1 Observational Transition Systems

We suppose that there exists a universal state space denoted by Υ and that each data type used in OTSs is provided. The data types include Bool for Boolean values. A data type is denoted by D with a subscript such as D_{o1} and D_o.

Definition 1 (OTSs). *An observational transition system (OTS) consists of*

- $\mathcal{O}$ *: A set of observers. Each observer is a function* $o : \Upsilon\ D_{o1} \ldots D_{om} \to D_o$*. The equivalence between two states* v_1, v_2 *(denoted as* $v_1 =_{\mathcal{S}} v_2$*) is defined with respect to (wrt) values returned by the observers.*
- $\mathcal{I}$ *: The set of initial states such that* $\mathcal{I} \subseteq \Upsilon$*.*
- $\mathcal{T}$ *: A set of transitions. Each transition is a function* $t : \Upsilon\ D_{t1} \ldots D_{tn} \to \Upsilon$*. Each transition* t*, together with any other parameters* $y_1, \ldots, y_n$*, preserves the equivalence between two states. Each* t *has the effective condition* $c\text{-}t : \Upsilon\ D_{t1} \ldots D_{tn} \to \text{Bool}$*. If* $\neg c\text{-}t(v, y_1, \ldots, y_n)$*, then* $t(v_1, y_1, \ldots, y_n) =_{\mathcal{S}} v$*.*

Given an OTS $\mathcal{S}$ and a state v, $t(v, y_1, \ldots, y_n)$ is called a successor state of v wrt $\mathcal{S}$ for any $y_1, \ldots, y_n$. We may omit "wrt $\mathcal{S}$" if it is clear from the context.

Definition 2 (Reachable States). *Given an OTS* $\mathcal{S}$ *and a set* $\mathbf{U}$ *of states, the reachable states from* $\mathbf{U}$ *wrt* $\mathcal{S}$ *are inductively defined as follows:*

- *Each state in* $\mathbf{U}$ *is reachable from* $\mathbf{U}$*.*
- *If a state* $v \in \Upsilon$ *is reachable from* $\mathbf{U}$*, so is* $t(v, y_1, \ldots, y_n)$ *for each* $t \in \mathcal{T}$ *and any other parameters* $y_1, \ldots, y_n$*.*

We may omit "wrt $\mathcal{S}$" and write "v is reachable from $\mathbf{U}$" if $\mathcal{S}$ is clear from the context. When $\mathbf{U}$ is $\mathcal{I}$, we may omit "from $\mathcal{I}$" and write "v is reachable wrt $\mathcal{S}$" or "v is reachable".

Let $\mathcal{R}_{\mathcal{S},\mathbf{U}}$ be the set of all states reachable from $\mathbf{U}$ wrt $\mathcal{S}$. $\mathcal{R}_{\mathcal{S},\mathbf{U}}$ may be called the state space reachable from $\mathbf{U}$. Let $\mathcal{R}_{\mathcal{S}}$ be $\mathcal{R}_{\mathcal{S},\mathcal{I}}$. $\mathcal{R}_{\mathcal{S}}$ may be called the reachable state space. When $\mathbf{U}$ is a singleton, say $\{u\}$, we may write $\mathcal{R}_{\mathcal{S},u}$, $v \in \mathcal{R}_{\mathcal{S},u}$ is called reachable from u and u is called backward-reachable from

$v \in \mathcal{R}_{\mathcal{S},u}$. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the depth from v_1 to v_2 wrt $\mathcal{S}$ ($\mathrm{depth}_{\mathcal{S}}(v_1, v_2)$) is 0 if $v_2 =_{\mathcal{S}} v_1$ and $d+1$ if $\mathrm{depth}_{\mathcal{S}}(v_1, v_3) = d$ and v_2 is a successor state of v_3. If v_2 is not reachable from v_1, $\mathrm{depth}_{\mathcal{S}}(v_1, v_2)$ is ∞. Let $\mathcal{R}^{\leq d}_{\mathcal{S},u}$ be $\{v \in \mathcal{R}_{\mathcal{S},u} \mid \mathrm{depth}_{\mathcal{S}}(u, v) \leq d\}$.

Definition 3 (Invariants). *Given an OTS $\mathcal{S}$, a state predicate $p : \Upsilon \to \mathrm{Bool}$ is an invariant wrt $\mathcal{S}$ if $(\forall v \in \mathcal{R}_{\mathcal{S}})\, p(v)$.*

We may omit "wrt $\mathcal{S}$" and write "p is an invariant" if $\mathcal{S}$ is clear from the context.

CafeOBJ, an algebraic specification language, is used to specify OTSs. Υ is denoted by a sort, say `Sys`. Each $o \in \mathcal{O}$ is denoted by an operator (called an observation operator) declared as follows: "`op o : Sys` $\mathtt{D}_{o1} \ldots \mathtt{D}_{om}$ `->` $\mathtt{D}_o$", where each $\mathtt{D}_*$ is a sort corresponding to D_*.

An arbitrary initial state in $\mathcal{I}$ is denoted by an operator declared as follows: "`op init : -> Sys {constr}`". Operators with no arguments such as `init` are called constants. For each $o \in \mathcal{O}$, declared is an equation "`eq o(init,`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) = f(`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) .`", where each $\mathtt{X}_*$ is a CafeOBJ variable of sort $\mathtt{D}_*$ and `f(`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`)` is a term denoting the value returned by o, together with any other parameters, in an arbitrary initial state. Note that each CafeOBJ variable occurring in an equation (or a transition rule; see §3) is universally quantified and its scope is in the equation (or the transition rule).

Each $t \in \mathcal{T}$ is denoted by an operator (called a transition operator) declared as follows: "`op t : Sys` $\mathtt{D}_{t1} \ldots \mathtt{D}_{tn}$ `-> Sys {constr}`". For each `o` and `t`, a conditional equation is declared: "`ceq o(t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`),`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) = o-t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`,`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`) if c-t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`) .`", where `c-t(S,...)` corresponds to $c\text{-}t(v, \ldots)$ and `o-t(S,...)` does not use any transition operators. The equation says how t changes the value observed by o if the effective condition holds. If `o-t(S,...)` is always the same as `o(S,`$\mathtt{X}_1$`,...,`$\mathtt{X}_m$`)`, the condition may be omitted.

For each `t`, one more conditional equation is declared: "`ceq t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`) = S if not c-t(S,`$\mathtt{Y}_1$`,...,`$\mathtt{Y}_n$`) .`", which says that t changes nothing if the effective condition does not hold.

As indicated by `{constr}`, `init` and each `t` are constructors of sort `Sys`[1]. They construct $\mathcal{R}_{\mathcal{S}}$.

A simple example is used to describe OTSs. The example used is a flawed mutual exclusion protocol.

Example 1 (Flawed Mutex Protocol). Multiple processes share a Boolean variable *locked* whose initial value is false. Each process executes the pseudo-program:

Loop: "Remainder Section"
  rs: **wait until** *locked* = false;
  es: *locked* := true;
    "Critical Section"
  cs: *locked* := false;

[1] Sort `Sys` denotes $\mathcal{R}_{\mathcal{S}}$ but not Υ if the constructor-based logic[11] is adopted, which is the current underlined logic of the OTS/CafeOBJ method.

Initially each process is in Remainder Section (RS). If a process wants to enter Critical Section (CS), it waits at label rs until *locked* becomes false and then sets *locked* to true at label es before entering CS. When it leaves CS, it sets *locked* to false at label cs and then goes back to RS.

How to specify an OTS $\mathcal{S}_{\mathrm{FMP}}$ formalizing the protocol is described. Two observers are used. The corresponding observation operators are as follows: "`op locked : Sys -> Bool`" and "`op pc : Sys Pid -> Label`", where sort `Pid` denotes process identifiers (IDs) and sort `Label` denotes the labels rs, es and cs. `locked` returns the value of *locked* in a given state, and `pc` returns the label at which a given process is in a given state.

In the rest of this section, let `S`, `I` and `J` be CafeOBJ variables of sorts `Sys`, `Pid` and `Pid`, respectively. The values returned by the two observers in an arbitrary initial state denoted by `init` are specified as follows: "`eq locked(init) = false .`" and "`eq pc(init,I) = rs .`".

Three transitions are used. The corresponding transition operators are as follows: "`ops try enter exit : Sys Pid -> Sys {constr}`". `try`, `enter`, and `exit` correspond to one iteration of the loop at label rs, the assignment at label es, and the assignment at label cs, respectively.

The set of equations specifying how `try` changes the values observed by the two observers is as follows:

```
eq locked(try(S,I)) = locked(S) .
ceq pc(try(S,I),J)
 = (if I = J then es else pc(S,J) fi) if c-try(S,I) .
ceq try(S,I) = S if not c-try(S,I) .
```

where `c-try(S,I)` is defined as `pc(S,I) = rs and not locked(S)`. `enter` and `exit` are defined likewise. Let `MUTEX` be a module in which $\mathcal{S}_{\mathrm{FMP}}$ is specified.

2.2 Falsification by Structural Induction

Verification of invariants is conducted by writing proof scores in CafeOBJ and executing them with the CafeOBJ system. Verification that a state predicate is an invariant wrt $\mathcal{S}_{\mathrm{FMP}}$ is used as an example to describe how to write proof scores in CafeOBJ. The state predicate used is $(\forall \mathtt{I}, \mathtt{J} : \mathtt{Pid})$ `inv1(S,I,J)`, where `inv1(S,I,J)` is `pc(S,I) = cs and pc(S,J) = cs implies I = J`. The predicate formalizes what is called the mutual exclusion property. Let `MUTEX-PREDS` be a module in which `MUTEX` is imported (namely that it is available) and state predicates to verify such as `inv1` are specified.

Verification starts with use of the structural induction on $\mathcal{R}_{\mathcal{S}_{\mathrm{FMP}}}$ (or sort `Sys`). Then, we have four CafeOBJ code fragments because there are the four constructors. Two out of the four CafeOBJ code fragments enclosed with commands `open` and `close` are as follows:

```
open MUTEX-BASE
  red inv1(init,i,j) .
close
```

```
open MUTEX-ISTEP
  eq s' = enter(s,k) .  red istep1 .
close
```

MUTEX-BASE is a module in which MUTEX-PREDS is imported. s, s', i, j and k are constants declared in MUTEX-BASE. s is used to denote an arbitrary state, s' an arbitrary successor state of s, and i, j and k arbitrary process identifiers. MUTEX-ISTEP is a module in which MUTEX-BASE is imported. istep1 is a constant declared in MUTEX-ISTEP. istep1 is defined as inv1(s,i,j) implies inv1(s',i,j). inv1(s',i,j) is the formula to prove in each induction case and inv1(s,i,j) is an instance of the induction hypothesis $(\forall \mathtt{I}, \mathtt{J} : \mathtt{Pid})$ inv1(s,I,J). Command open makes a temporary module in which a given module is imported, and command close destroys such a temporary module. Command red reduces a given term by regarding equations as left-to-right rewrite rules. The four CafeOBJ code fragments are the proof score in progress of inv1. CafeOBJ code fragments in proof scores (in progress as well) are called proof passages. The proof passage for init is for the base case, while the remaining three ones for the induction step, or the three induction cases.

If CafeOBJ returns true for a proof passage, the proof passage is discharged. If CafeOBJ returns true for each proof passage in the proof score of a predicate and all lemmas used in the proof score have been proved, the predicate has been proved, namely that it is an invariant wrt an OTS concerned.

CafeOBJ returns true for the proof passage for init and then the base case is discharged. Since CafeOBJ does not return true for the remaining three, however, we need to transform the proof passages with case splitting and lemma conjecture/use.

Let us take the induction case for enter. The proof passage is first transformed into two proof passages with case splitting based on the effective condition of enter. The two proof passages correspond to the two cases: (1) c-enter(s,k) = false, and (2) c-enter(s,k) = true. CafeOBJ returns true for the first case but not for the second case. Since c-enter(s,k) = true is equivalent to pc(s,k) = es, the latter can be used instead of the former. Even if so, CafeOBJ does not return true for the second case.

The proof passage is next transformed into four proof passages with case splitting based on the two propositions i = k and j = k found in the result returned by CafeOBJ. The four proof passages correspond to the four cases: (1) i = k, j = k, (2) (i = k) = false, (j = k) = false, (3) i = k, (j = k) = false, and (4) (i = k) = false, j = k. CafeOBJ returns true for the first two cases but not for the remaining two cases. Let us take the third case.

The corresponding proof passage is then transformed into two proof passages with case splitting based on the proposition pc(s,j) = cs. The two proof passages correspond to the two cases: (1) (pc(s,j) = cs) = false, and (2) pc(s,j) = cs. CafeOBJ returns true for the first case but false for the second case.

The proof passage corresponding to the second case is as follows:

```
open MUTEX-ISTEP
 eq pc(s,k) = es .    eq i = k .    eq (j = k) = false .
 eq pc(s,j) = cs .    eq s' = enter(s,k) .    red istep1 .
close
```

If `inv1` holds for $\mathcal{S}_{\mathrm{FMP}}$, then an arbitrary state `s` characterized by the first four equations in the proof passage is unreachable wrt $\mathcal{S}_{\mathrm{FMP}}$. Therefore, we can conjecture a lemma from the four equations to discharge the proof passage. If one of such equations such as `i = k` has a fresh constant as one side and CafeOBJ still returns false even after replacing all the occurrences of the fresh constant with the other side in the proof passage and deleting the equation, then we can use the remaining equations to conjecture a lemma.

For this proof passage, a lemma can be conjectured from the following three equations by basically conjoining the equations with conjunctions, negating the obtained formula, and replacing fresh constants with appropriate variables: `eq pc(s,i) = es .`, `eq (j = i) = false .`, and `eq pc(s,j) = cs .` The lemma is `not(pc(S,I) = es and pc(S,J) = cs and not(I = J))`, which is referred as `inv2(S,I,J)`. This lemmas has the property that if `inv1` holds for $\mathcal{S}_{\mathrm{FMP}}$, so does `inv2`. Or in contrapositive form, if `inv2` does not hold for $\mathcal{S}_{\mathrm{FMP}}$, neither does `inv1`. Lemmas that have this property are called necessary lemmas of the original predicates[4]. `inv2` is a necessary lemma of `inv1` and only the lemma needed to discharge the proof score of `inv1`.

In the verification of `inv2`, we conjecture the two lemmas `not(pc(S,I) = rs and pc(S,J) = cs and not(I = J) and not(locked(S)))` and `not(pc(S,I) = es and pc(S,J) = es and not(I = J))`, which are referred as `inv3(S,I,J)` and `inv4(S,I,J)`, respectively. Both `inv3` and `inv4` are necessary lemmas of `inv2`.

We only need `inv1` as a lemma to discharge the proof score of `inv3`, but conjecture the following lemma for `inv4`: `not(pc(S,I) = es and pc(S,J) = rs and not(I = J) and not(locked(S)))`, which is referred as `inv5(S,I,J)`. `inv5` is a necessary lemma of `inv4`.

In the verification that `inv5` holds for $\mathcal{S}_{\mathrm{FMP}}$, the following lemma is conjectured: `not(pc(S,I) = rs and pc(S,J) = rs and not(I = J) and not(locked(S)))`, which is referred as `inv6(S,I,J)`. `inv6` is a necessary lemma of `inv5`.

`inv6(init,i,j)` reduces to `false` if `i` is different from `j`, from which we can conclude that `inv1` does not hold for $\mathcal{S}_{\mathrm{FMP}}$ because every lemma used is a necessary lemma of its original predicate. This example demonstrates that structural induction can also be used to falsify that a system enjoys an invariant.

3 Bounded Model Checking (BMC) of OTSs

Instead of a set of equations, a transition rule can also be used to specify each transition $t \in \mathcal{T}$ of an OTS $\mathcal{S}$. If so, the search functionality can be used. The search functionality is in the form:

`red` *init* `=(`n`,`d`)=>*` *pattern* `suchThat` *cond* `.`

where *init* is a ground term, *pattern* a state pattern, *cond* a Boolean term, and n and d natural numbers or `*` denoting the infinity. "suchThat *cond*" is an option. The search functionality exhaustively traverses $\mathcal{R}^{\leq d}_{\mathcal{S},init}$ in a breadth-first manner

so as to find at most n states such that they match *pattern* and satisfy *cond*. When *init* is an initial state of $\mathcal{S}$ and the negation of a state predicate concerned is expressed in *pattern* and *cond*, the search functionality conducts BMC of an invariant, namely that it exhaustively traverses $\mathcal{R}^{\leq d}_{\mathcal{S},init}$ to find a counterexample showing that the state predicate is not an invariant.

$\mathcal{S}_{\mathrm{FMP}}$ is used as an example to describe how to specify transitions in transition rules. To specify transitions in transition rules, it is necessary to design the configuration of states. Associative-commutative collections of values observed by observers can be used as the configuration. For the configuration, the following are declared: "`op void : -> Sys {constr}`" and "`op __: Sys Sys -> Sys {constr assoc comm id: void}`". `Sys` is the sort denoting states, which are constructed with `void` and the juxtaposition operator. The juxtaposition operator is associative, commutative, and has `void` as its identity.

Since $\mathcal{S}_{\mathrm{FMP}}$ has two observers, the following two operators that hold two kinds of values observed by the two observers are declared: "`op (pc[_]:_) : Pid Label -> Obs {constr}`" and "`op locked:_: Bool -> Obs {constr}`". `Obs` is a subsort of `Sys`. Therefore, a collection of terms whose sorts are `Obs` denotes a state.

If two processes `p1` and `p2` participate in the protocol, the initial state (denoted by `init`) is expressed as follows: "`eq init = (pc[p1]: rs) (pc[p2]: rs) (locked: false) .`".

Let S, I, J, L1, L2 and B be CafeOBJ variables of sorts `Sys`, `Pid`, `Pid`, `Label`, `Label` and `Bool`, respectively, in the rest of the section. The three transitions are specified in transition rules as follows:

```
trans [try] : (pc[I]: rs) (locked: false)
  => (pc[I]: es) (locked: false) .
trans [enter] : (pc[I]: es) (locked: B)
  => (pc[I]: cs) (locked: true) .
trans [exit] : (pc[I]: cs) (locked: B)
  => (pc[I]: rs) (locked: false) .
```

where `try`, `enter` and `exit` enclosed with "`[`" and "`]`" are the labels (names) of the three transition rules, respectively.

The following command (the search functionality) can be used to try to find a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property: "`red init =(1,*)=>* (pc[I]: L1) (pc[J]: L2) S suchThat (not (L1 == cs and L2 == cs implies I == J)) .`". The command can be equivalently transformed into "`red init =(1,*)=>* (pc[I]: cs) (pc[J]: cs) S .`". Each of the commands can find a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property.

Since a state in which `inv1` does not hold is located at depth 4 from the initial state, the following command does not find the counterexample: "`red init =(1,3)=>* (pc[I]: cs) (pc[J]: cs) S .`".

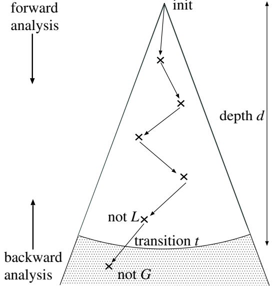


Fig. 1. Forward & backward reachability analysis

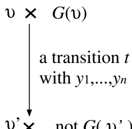


Fig. 2. A situation requesting a lemma in induction

4 Forward & Backward Reachability Analysis

Our primary goal is to discover a counterexample showing that a state predicate is not an invariant wrt an OTS $\mathcal{S}$.

4.1 Forward Reachability Analysis

Forward reachability analysis is to start with initial states and traverse the reachable state space to find some states in which some conditions hold (see Figure 1). Model checking, especially BMC, is a typical forward reachability analysis method. The search functionality is also a forward reachability analysis method. The method is fascinating as well as powerful in that it can fully automatically discover a counterexample showing that a state predicate is not an invariant. This is how we have found a counterexample showing that $\mathcal{S}_{\mathrm{FMP}}$ does not enjoy the mutual exclusion property in §3.

4.2 Backward Reachability Analysis

Backward reachability analysis method is to start with some states $\upsilon_1, \ldots, \upsilon_n$ (which may or may not be reachable) such that a state predicate does not hold and traverse the state space backward-reachable from $\upsilon_1, \ldots, \upsilon_n$ to check if an initial state of $\mathcal{S}$ is backward-reachable from υ_i for some $i \in \{1, \ldots, n\}$ (see Figure 1). If an initial state of $\mathcal{S}$ is backward-reachable from υ_i, then υ_i is reachable and then the predicate is not an invariant. If any initial state is not backward-reachable from an arbitrary state in which the predicate does not hold, we can conclude that the predicate is an invariant.

Structural induction can be regarded as a backward reachability analysis method. Let us consider an induction case for a transition t, together with $y_1, \ldots, y_n$. Let υ' be $t(\upsilon, y_1, \ldots, y_n)$ for an arbitrary state υ and G be a state predicate concerned. If $G(\upsilon)$ and $\neg G(\upsilon')$ (see Figure 2), then all we are concerned with is whether υ is reachable. If it is, G is not an invariant. Otherwise,

this induction case is discharged. To this end, what we can do is to conjecture a lemma. Although we do not know $\text{depth}_{\mathcal{S}}(\mathit{init}, \upsilon)$ for some initial state *init* nor whether υ is reachable, it is true that υ is backward-reachable from υ'. That is to say, one step is taken back from a state such that G does not hold by structural induction. This is how we have found a counterexample showing that $\mathcal{S}_{\text{FMP}}$ does not enjoy the mutual exclusion property in §2.2.

4.3 Combination

Both forward and backward reachability analysis methods have the pros and cons. The search functionality can fully automatically discover a counterexample showing that a state predicate is not an invariant. This is, however, only the case when a state in which the predicate does not hold is located at a position that is not far from a given initial state *init*. The distance to a state υ from *init* is not simply $\text{depth}_{\mathcal{S}}(\mathit{init}, \upsilon)$. Let d be $\text{depth}_{\mathcal{S}}(\mathit{init}, \upsilon)$ and then the distance crucially depends on the number of states in $\mathcal{R}^{\leq d}_{\mathcal{S},\mathit{init}}$.

If the reachable state space is huge or unbounded, there exists an upper bound d such that $\mathcal{R}^{\leq d}_{\mathcal{S},\mathit{init}}$ can be exhaustively traversed but $\mathcal{R}^{\leq d+1}_{\mathcal{S},\mathit{init}}$ cannot. If that is the case, the search functionality may not discover any counterexamples even though there exist some (see Figure 1). This is due to the notorious state explosion problem. If $\mathcal{R}^{\leq 4}_{\mathcal{S}_{\text{FMP}},\texttt{init}}$ was too large, the search functionality would not find a counterexample showing that the protocol does not enjoy the mutual exclusion property.

As described in §2.2, structural induction can be used to find a counterexample showing that a state predicate is not an invariant wrt an OTS $\mathcal{S}$. Generally, however, we need to conjecture a lot of necessary lemmas to have one such that it does not hold for some initial states. It also costs more than the search functionality.

But, structural induction may alleviate the state explosion problem, which bothers the search functionality. If $\mathcal{R}^{\leq 4}_{\mathcal{S}_{\text{FMP}},\texttt{init}}$ was too large, we could try to find a counterexample for `inv2`, which is a necessary lemma of `inv1`. The following command finds a counterexample: "`red init =(1,3)=>* (pc[I]: es) (pc[J]: cs) S .`". The command lets us know the reachable state `(pc[p2]: es) (pc[p1]: cs) (locked: true)` in which `inv2` does not hold. Consequently, `inv1` is not an invariant wrt $\mathcal{S}_{\text{FMP}}$, either.

This is how we have come up with one possible way to complement each other, which is called induction-guided falsification (IGF)[4]. IGF is a combination of the search functionality (or BMC) and structural induction, but can be regarded as a combination of forward and backward reachability analysis methods because structural induction can be regarded as a backward reachability analysis method. If we exactly obey IGF, namely that every lemma conjectured is a necessary lemma, then once you find a counterexample for a lemma, you can quickly conclude that the original state predicate is not invariant. Even if non-necessary lemmas are used, the basic concept behind IGF, namely a combination of forward and backward reachability analysis methods, can be used. Non-necessary lemmas are less complicated than necessary lemmas.

Given an OTS $\mathcal{S}$ and a state predicate p, let $\mathcal{L}_{\mathcal{S},p}$ be a set of lemmas that can discharge the proof score that p is an invariant wrt $\mathcal{S}$. A variant of IGF is as follows:

Input: an OTS $\mathcal{S}$, a state predicate p, a natural number d;
Output: Verified or Falsified that p is an invariant wrt $\mathcal{S}$;
1. $\mathcal{P} := \text{enqueue}(\text{empty-queue}, p)$ and $\mathcal{Q} := \emptyset$.
2. Repeat the following until $\mathcal{P} = \text{empty-queue}$.
3. $q := \text{top}(\mathcal{P})$ and $\mathcal{P} := \text{dequeue}(\mathcal{P})$.
4. Search $R_{\mathcal{S}}^{\leq d}$ for a state v such that $\neg q(v)$.
   If such a state is not found, go to 8.
5. Search $R_{\mathcal{S},v}^{\leq d}$ for a state such that $\neg p(v)$.
   If such a state is found, terminate and return Falsified.
6. Search $R_{\mathcal{S},v}^{\leq d}$ for a state v such that $\neg\text{main}_q(v)$, where main_q is a state predicate in $\mathcal{Q}$, for which q is used as a lemma.
   If such a state is found, $q := \text{main}_q$, delete q and the state predicates that are used as lemmas only for q from $\mathcal{P}$ and $\mathcal{Q}$ and go to 5.
7. Find a lemma q' of main_q such that $q \Rightarrow q'$ and q' is not equivalent to q, $q := q'$ and go to 4.
8. Compute $\mathcal{L}_{\mathcal{S},q}$ by structural induction on $\mathcal{R}_{\mathcal{S}}$.
9. $\mathcal{Q} := \mathcal{Q} \cup \{q\}$ and enqueue each in $\mathcal{L}_{\mathcal{S},q} - (\mathcal{Q} \cup \text{q2s}(\mathcal{P}))$ into $\mathcal{P}$, where $\text{q2s}(\mathcal{P})$ is the set that consists of the elements of $\mathcal{P}$.
10. Terminate and return Verified.

For example, if `inv2` was not a necessary lemma of `inv1`, the following command would find a counterexample for `inv1`: "`red (pc[p2]: es) (pc[p1]: cs) (locked: true) =(1,3)=>* (pc[I]: cs) (pc[J]: cs) S .`".

5 Application of the Variant of IGF to NSPK

5.1 NSPK and Agreement Property

NSPK[7] can be described as the three message exchanges:

Init: $p \rightarrow q$ $\{n_p, p\}_{k(q)}$
Resp: $q \rightarrow p$ $\{n_p, n_q\}_{k(p)}$
Ack: $p \rightarrow q$ $\{n_q\}_{k(q)}$

Each principal such as p and q is given a pair of keys (public and private keys). $\{m\}_{k(x)}$ is the ciphertext obtained by encrypting m with the principal x's public key. n_x is a nonce generated by a principal x.

The agreement property is as follows. Whenever a protocol run has been successfully completed by p and q,

AP1 the principal that p is communicating with is really q, and
AP2 the principal that q is communicating with is really p.

5.2 Specification for Structural Induction

We use the standard assumptions for protocol verification. Among them are that the cryptsystem used is perfect and the behaviors of malicious principals are formalized by the Dolev-Yao intruder[12]. Since we are only interested in invariants, it is not necessary to consider blocking of messages by the intruder.

A nonce generated by p for sending it to q is denoted by a term $\mathtt{n}(p, q, r)$ whose sort is `Nonce`, where r is a random number making the nonce unguessable and unique. Our formalism of NSPK allows a principal to participate in multiple sessions simultaneously. For each session, a principal needs to generate a fresh nonce.

Ciphertexts $\{n_p, p\}_{k(q)}$, $\{n_p, n_q\}_{k(p)}$ and $\{n_q\}_{k(q)}$ used in Init, Resp and Ack messages, respectively, are denoted by terms $\mathtt{enc1}(q, n_p, p)$, $\mathtt{enc2}(p, n_p, n_q)$ and $\mathtt{enc3}(q, n_q)$, respectively. Their sorts are `Cipher`i for $i = \mathtt{1}, \mathtt{2}, \mathtt{3}$, respectively.

Init, Resp and Ack messages are denoted by terms $\mathtt{m}i(p?, p, q, e_i)$ for $i = \mathtt{1}, \mathtt{2}, \mathtt{3}$, respectively. Their sorts are `Message`i for $i = \mathtt{1}, \mathtt{2}, \mathtt{3}$, respectively. The first argument $p?$ is a creator (an actual sender) of the message, the second argument p a seeming sender, the third argument q a receiver and the fourth argument e_i a ciphertext. The first argument is meta-information in that when q receives $\mathtt{m}i(p?, p, q, e_i)$, q cannot loot at $p?$. If $p?$ is different from p, then $p?$ is the intruder and the message has been faked by the intruder.

The network is formalized as an associative-commutative collection of messages whose sort is `Network`. Associative-commutative collections may be called just collections. A constant `empty` and a juxtaposition operator are the constructors of collections of not only messages but also the others such as nonces. We suppose that once a message $\mathtt{m}i(p?, p, q, e_i)$ is put into the network, it will be never deleted, and if there exists such a message in the network, q can receive it. When q has received it, q thinks that it originates in p.

We formalize the behaviors of NSPK as an OTS $\mathcal{S}_{\mathrm{NSPK}}$. We use three observers that are denoted by the observation operators: "`op network : System -> Network`", "`op rands : System -> RandSoup`", and "`op nonces : System -> NonceSoup`", where `System` is a sort denoting the (reachable) state space, `RandSoup` a sort denoting collections of random numbers, and `NonceSoup` a sort denoting collections of nonces. Given a state s, $\mathtt{network}(s)$ returns the network, the collections of messages that haven been sent up to s, $\mathtt{rands}(s)$ the collection of (old) random numbers that have been used up to s, and $\mathtt{nonces}(s)$ the collection of nonces that have been gleaned by the intruder up to s.

An arbitrary initial state is denoted by a constant `init` whose sort is `System`. `init` is a constructor of `System`. The three observation operators return `empty` for `init`.

Three transitions are used to formalize sending Init, Resp and Ack messages exactly obeying the protocol, respectively. The corresponding transition operators are as follows: "`op sdm1 : System Principal Principal Random -> System {constr}`", "`op sdm2 : System Principal Principal Principal Random Nonce -> System {constr}`", and "`op sdm3 : System Principal Principal Principal Nonce Nonce -> System {constr}`".

The set of equations defining sdm2 is as follows:

```
ceq network(sdm2(S,Q?,P,Q,R,N)) = m2(P,P,Q,enc2(Q,N,n(P,Q,R)))
 network(S)  if c-sdm2(S,Q?,P,Q,R,N) .
ceq rands(sdm2(S,Q?,P,Q,R,N)) = R rands(S) if c-sdm1(S,P,Q,R) .
ceq nonces(sdm2(S,Q?,P,Q,R,N)) = (if Q = intruder then N n(P,Q,R)
 nonces(S) else nonces(S) fi)  if c-sdm2(S,Q?,P,Q,R,N) .
ceq sdm2(S,Q?,P,Q,R,N) = S if not c-sdm2(S,Q?,P,Q,R,N) .
```

where `c-sdm2(S,Q?,P,Q,R,N)` is `m1(Q?,Q,P,enc1(P,N,Q)) \in network(S) and not(R \in rands(S))`.

The remaining two transition operators can be defined likewise. Symbols that appear in terms and are composed of capitals, numerals and `?` are CafeOBJ variables in this section. Among them are `S`, `Q?` and `RS2`. Theirs sorts can be understood from the context.

`c-sdm2(S,Q?,P,Q,R,N)` says that there exists an Init message that seems to have been sent to `Q` by `P` in the network and `R` is a fresh random number. If that is the case, `Q` can receive the message and finds that the message obeys the protocol. Then, the Resp message `m2(P,P,Q,...)` is put into the network as the reply to the Init message. Since `R` is used in `m2(P,P,Q,...)`, it is put into the collection of old random numbers. If `Q` is the intruder, the intruder can decrypt the ciphertext in `m2(P,P,Q,...)` and obtain the two nonces in it, which are put into the collection of nonces. Otherwise, the collection of nonces does not change. If `c-sdm2(S,Q?,P,Q,R,N)` does not hold. nothing changes. Receiving messages are implicitly formalized in transition operators.

Two kinds of values can be used to fake messages: messages and nonces. Since there are three kinds of messages, we use six transitions to formalize faking messages based on the gleaned information by the intruder. Due to the space limitation, we only describe two transitions faking Resp messages based on messages and nonces, respectively. The corresponding transition operators are as follows: "`op fkm21 : System Principal Principal Message2 -> System {constr}`" and "`op fkm22 : System Principal Principal Nonce Nonce -> System {constr}`.

The remaining four transition operators can be declared likewise.

The set of equations defining fkm22 is as follows:

```
ceq network(fkm22(S,P,Q,N1,N2))  = m2(intruder,P,Q,enc2(Q,N1,N2))
   network(S)  if c-fkm22(S,P,Q,N1,N2) .
eq rands(fkm22(S,P,Q,N1,N2)) = rands(S) .
eq nonces(fkm22(S,P,Q,N1,N2)) = nonces(S) .
ceq fkm22(S,P,Q,N1,N2) = S if not c-fkm22(S,P,Q,N1,N2) .
```

where `c-fkm22(S,P,Q,N1,N2)` is `N1 \in nonces(S) and N2 \in nonces(S) and not(N1 = N2)`. `c-fkm22(S,P,Q,N1,N2)` says that the intruder has gleaned two different nonces. If that is the case, the intruder can fake a Resp message `m2(intruder,P,Q,...)`, which is put into the network. Otherwise, nothing changes.

`fkm21` and the remaining four transition operators can be defined likewise.

5.3 Specification for Search

Since there are the three observers, the following three operators are used to hold the values observed by them: "`op network:_: Network -> Obs {constr}`", "`op rands:_: RandSoup -> Obs {constr}`", and "`op nonces:_: NonceSoup -> Obs {constr}`". In addition to them, two more operators are used to hold two values: "`op prins:_: PrinSoup -> Obs {constr}`" and "`op rands2:_: RandSoup -> Obs {constr}`", where `PrinSoup` is a sort denoting collections of principals. The first operator holds a collection of principals participating in the protocol and the second a collection of fresh random numbers that can be used in the protocol. The two values are not modified by any transitions.

We suppose that three principals including the intruder participate in the protocol and two fresh random numbers are available. Then, the initial state denoted by `init` is expressed as "`(network: empty) (rands: empty) (nonces: empty) (prins: (p q intruder)) (rands2: (r1 r2))`".

The transition denoted by `sdm2` is specified in the following transition rule:

```
ctrans [sdm2] : (network: (m1(Q?,Q,P,enc1(P,N,Q)) NW))
 (rands: RS) (nonces: NS) (rands2: (R RS2))
 => (network: (m2(P,P,Q,enc2(Q,N,n(P,Q,R)))
              m1(Q?,Q,P,enc1(P,N,Q)) NW))
 (rands: (R RS))
 (nonces: (if Q == intruder then N n(P,Q,R) NS else NS fi))
 (rands2: (R RS2))  if not(R \in RS) .
```

The transition denoted by `fkm22` is specified in the following transition rule:

```
trans [fkm22] :
 (network: NW) (nonces: (N1 N2 NS)) (prins: (P Q PS))
 => (network: (m2(intruder,P,Q,enc2(Q,N1,N2)) NW))
 (nonces: (N1 N2 NS)) (prins: (P Q PS)) .
```

The remaining seven transitions can be specified likewise.

5.4 Falsification

AP2 is formalized in terms of the following state predicate `inv2`:

```
eq inv2(S,P,Q,P?,R,N) = (not(Q = intruder) and
 m2(Q,Q,P,enc2(P,N,n(Q,P,R))) \in network(S) and
 m3(P?,P,Q,enc3(Q,n(Q,P,R))) \in network(S)
 implies m3(P,P,Q,enc3(Q,n(Q,P,R))) \in network(S)) .
```

AP1 can also be formalized likewise.

What we did first is to find an upper bound d such that $\mathcal{R}^{\leq d}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$ can be exhaustively traversed as follows: "`red init =(1,5)=>* S suchThat false .`". On a laptop with 2.33GH CPU and 3GB RAM, 5 was the upper bound[2].

[2] Since the implementation of the CafeOBJ search functionality was not matured enough, Maude was used to conduct the experiment described in this section.

In $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$, no counterexample was discovered for `inv1` and `inv2`. The following command tries to find a counterexample for `inv2`:

```
red init =(1,5)=>* (network: (m2(Q,Q,P,enc2(P,N,n(Q,P,R)))
 m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) S
 suchThat (not(not(Q == intruder) implies
 m3(P,P,Q,enc3(Q,n(Q,P,R))) \in m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) .
```

Then, we conjectured lemmas to discharge the proof scores of `inv1` and `inv2`. Five lemmas were conjectured. Two out of them are as follows:

```
eq inv4(S,P,Q,N,R,M2) = (not(P = intruder) and not(Q = intruder)
  and m1(P,P,Q,enc1(Q,n(P,Q,R),P)) \in network(S) and
  M2 \in network(S) and cipher2(M2) = enc2(P,n(P,Q,R),N)
  implies m2(Q,Q,P,enc2(P,n(P,Q,R),N)) \in network(S)) .
eq inv5(S,N) = (N \in nonces(S)
  implies creator(N) = intruder or forwhom(N) = intruder) .
```

Each of the five lemmas is a necessary one of neither `inv1` not `inv2`.

No counterexample was found in $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$ for the four lemmas including `inv4`. But, a counterexample was found for `inv5`, which formalizes what is called the nonce secrecy property. Hence, NSPK does not enjoy the nonce secrecy property. Since `inv5` is a necessary lemma of neither `inv1` nor `inv2`, however, we cannot immediately conclude that NSPK does not enjoy the agreement property.

The state in which `inv5` does not hold is as follows:

```
eq s115890 = (nonces: (n(q,p,r2) n(p,intruder,r1)))
  (network: ( m1(intruder,p,q,enc1(q,n(p,intruder,r1),p))
   m1(p,p,intruder,enc1(intruder,n(p,intruder,r1),p))
   m2(intruder,intruder,p,enc2(p,n(p,intruder,r1),n(q,p,r2)))
   m2(q,q,p,enc2(p,n(p,intruder,r1),n(q,p,r2)))
   m3(p,p,intruder,enc3(intruder,n(q,p,r2)))))
  (rands: (r1 r2)) (prins: (intruder p q)) (rands2: (r1 r2)) .
```

The state, which is reachable, is reported by the search functionality. This is the 115890th state that the search functionality has visited from `init`.

Instead of $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{init}}$, we can then traverse $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{s115890}}$ to find a counterexample for `inv1` and `inv2`. But, $\mathcal{R}^{\leq 5}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{s115890}}$ was too large to be exhaustively traversed. Therefore, we traversed $\mathcal{R}^{\leq 4}_{\mathcal{S}_{\mathrm{NSPK}},\mathtt{s115890}}$ to find a counterexample for `inv1` and `inv2`. The command to find a counterexample for `inv2` is as follows:

```
red s115890 =(1,4)=>* (network: (m2(Q,Q,P,enc2(P,N,n(Q,P,R)))
 m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) S
 suchThat (not(not(Q == intruder) implies
 m3(P,P,Q,enc3(Q,n(Q,P,R))) \in m3(P?,P,Q,enc3(Q,n(Q,P,R))) NW)) .
```

No counterexample was found for `inv1` but a counterexample was found for `inv2`. The counterexample found is the same as that was found by Lowe[13].

6 Related Work

Another possible combination of BMC and structural induction has been proposed: k-induction[14]. It has been implemented in SAL[15], which is a toolkit for analyzing state machines. The primary purpose of k-induction is verification. k-induction can be used to verify that a system (formalized as a state machine) enjoys an invariant. It is, however, necessary to fix the number of entities (such as processes) participating in a system. Since standard structural induction is used in (the variant of) IGF, only one step is taken back from a state in which a state predicate concerned does not hold. k-induction allows to take more than one step back from such a state. Hence, it may make (the variant of) IGF more powerful to adopt k-induction, which is one piece of our future work.

Maude-NPA[3] has been implemented in Maude[6], relying on the narrowing search functionality. While the term *init* should be ground in the ordinary search functionality, it can contain variables in the narrowing search functionality. Hence, *init* can express an arbitrary state in which a state predicate concerned does not hold. The (ordinary and narrowing) search functionality can conduct a backward reachability analysis by reversing the transition rule specifying each transition. This is how Maude-NPA conducts a backward reachability analysis. The backward reachability analysis method used by Maude-NPA may be used for (the variant of) IGF. If so, we only need to have one type of specifications in which transitions are specified in transition rules. This is another piece of our future work. The narrowing search functionality may be used to implement more general k-induction such that it is not necessary to fix number of entities participating in a system. This is yet another piece of our future work.

7 Conclusion

The primary purpose of (the variant of) IGF is to falsify that a system enjoys a property. The mainly used technique for this purpose is testing, which can be roughly classified into exhaustive and non-exhaustive testing. (Bounded) Model checking can be used for the former. Daniel Jackson, who is the main designer of Alloy[2], has formed the small scope hypothesis, which says that most errors can be found by testing a program for all test inputs within some small scope[16]. This implies that it is more beneficial to exhaustively test a program within some small scope than to test it for some randomly generated test cases within larger scope. This is why Alloy has adopted a SAT-based bounded model checker.

The state in which AP2 (`inv2`) does not hold has not been found within the small scope such that the search functionality can exhaustively traverse the scope. Some may suggest that the nonce secrecy property should be taken into account instead of the agreement property because the former is more fundamental than the latter for authentication protocols. This is why almost all analyzes of NSPK have taken into account the nonce secrecy property[8,9,3]. Generally, however, we do not know in advance what is more fundamental than a property concerned such as the agreement property for a system such as NSPK. Therefore, we need to extend the scope that can be exhaustively traversed so as to find

more errors. This is why (the variant of) IGF has been proposed and a backup case study has been conducted.

One piece of our future work is to design and implement a tool supporting (the variant of) IGF. We may use the translator[17] from state machine specifications in CafeOBJ into those in Maude and the technique to discover lemmas used in Crème[18], an automatic invariant prover for state machine specifications in CafeOBJ.

References

1. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
2. Jackson, D.: Alloy: A lightweight object modeling notation. ACM TOSEM 11, 256–290 (2002)
3. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyser and its meta-logical properties. TCS 367, 162–202 (2006)
4. Ogata, K., Nakano, M., Kong, W., Futatsugi, K.: Induction-guided falsification. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 114–131. Springer, Heidelberg (2006)
5. Diaconescu, R., Futatsugi, K.: CafeOBJ report. AMAST Series in Computing, vol. 6. World Scientific, Singapore (1998)
6. 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. LNCS, vol. 4350. Springer, Heidelberg (2007)
7. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. CACM 21, 993–999 (1978)
8. Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In: Margaria, T., Steffen, B. (eds.) TACAS 1996. LNCS, vol. 1055, pp. 147–166. Springer, Heidelberg (1996)
9. Denker, G., Meseguer, J., Talcott, C.: Protocol specification and analysis in Maude. In: Workshop on Formal Methods and Security Protocols (1998)
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. Gâinâ, D., Futatsugi, K., Ogata, K.: Constructor-based institutions. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 398–412. Springer, Heidelberg (2009)
12. Dolev, D., Yao, A.C.: On the security of public key protocols. IEEE TIT IT-29, 198–208 (1983)
13. Lowe, G.: An attack on the Needham-Schroeder public-key authentication protocol. IPL 56, 131–133 (1995)
14. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: CAV 2003. LNCS, vol. 2392, pp. 14–26. Springer, Heidelberg (2003)

15. 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)
16. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press, Cambridge (2006)
17. Zhang, M., Ogata, K., Nakamura, M.: Specification translation of state machines from equational theories into rewrite theories. In: Zhu, H. (ed.) ICFEM 2010. LNCS, vol. 6447, pp. 678–693. Springer, Heidelberg (2010)
18. Nakano, M., Ogata, K., Nakamura, M., Futatsugi, K.: Crème: An automatic invariant prover of behavioral specifications. IJSEKE 17, 783–804 (2007)

Model Checking a Model Checker: A Code Contract Combined Approach*

Jun Sun[1], Yang Liu[2], and Bin Cheng[2]

[1] Singapore University of Technology and Design
sunjun@sutd.edu.sg
[2] School of Computing, National University of Singapore
{liuyang,chenbing}@comp.nus.edu.sg

Abstract. Model checkers, like any complex software, are subject to bugs. Unlike ordinary software, model checkers are often used to verify safety critical systems. Their correctness is thus vital. Verifying model checkers is extremely challenging because they are always complicated in logic and highly optimized. In this work, we propose a code contract combined approach for checking model checkers and apply it to a home-grown model checker PAT. In this approach, we firstly embed programming contracts (i.e., pre/post-conditions and invariants) into its source code, which can capture correctness of model checking algorithms, underlying data structures, consistency between different model checking parameters, etc. Then, interface models of complicated data structures and graphical user interfaces (GUI) are built and model checked. By linking the interface models with actual source codes and exhausting all execution sequences of interface models using PAT, *we model check PAT using itself*! Our experience shows that the approach is effective in identifying common bugs or subtle flaws that result from extremely improbable events.

1 Introduction

After two decades of development, model checking [8] has emerged as an effective method for verification of critical systems. It has established as a system validation method complementing standard techniques like simulation and testing. There have been a number of recent successful stories. The static driver verifier which uses the SLAM verification engine has been reported to find many driver model violations [1]. In 2009, it was reported that model checking has been used to replace testing in Intel CoreTM i7 processor (with millions of registers) execution engine validation [17].

Model checkers, like any non-trivial software, are subject to bugs. This is evidenced by the bug collection for established model checkers like SPIN [14] and NuSMV [7]. Model checkers are, nonetheless, distinguished from ordinary software due to their very nature. Firstly, they are always complicated in computational logic. Many complicated model checking algorithms, in the name of efficiency, have been proposed to verify a variety of system properties. Furthermore, sophisticated state reduction techniques are

* This research was partially supported by a grant "SRG ISTD 2010 001" from Singapore University of Technology and Design.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 518–533, 2010.
© Springer-Verlag Berlin Heidelberg 2010

often applied, for instance, partial order reduction [22], symmetric reduction [10], data abstraction [2], or their combinations. Secondly, because efficiency is essential, model checkers are often highly optimized, which implies that they may not be designed for rigorous system maintenance or testing. Lastly, because model checking techniques are developing rapidly, model checkers are often updated frequently. This is evidenced by the update history of popular model checkers like SPIN, Uppaal, and so on.

Model checkers are often applied to safety critical systems. If a property is falsified, correctness of the model checker can be validated by checking whether the counterexample is a real one. It is when a property is claimed true - which is often the last act of formal verification, the correctness of the model checker must be assumed in order to trust the verification result. Given the importance of model checkers, it is essential to verify them formally or at least develop ways to systematically improve their quality.

There are multiple candidate approaches. First, theorem proving can be used to prove the correctness of model checking *algorithms*. It is, however, unpractical in verifying model checkers. The second candidate is push-button software verification technique like model checking. Still, completely verifying model checkers is beyond the capability of state-of-art model checking solutions. One important factor is their size. For instance, NuSMV has about 180 KLOC and SPIN has more than 30 KLOC. The current software verification tools can handle programs (often from constrained scenarios) up to tens of KLOC and often require manual simplifications of the code under analysis [21]. Furthermore, state-of-art software verification (e.g., the SLAM project [1]) relies on building an abstract finite state model from a program using predicate abstraction, which is highly non-trivial and expensive. Given that model checkers are often updated frequently, this approach is unpractical.

Contribution. In this work, we take the challenge of systematically validating a real-world model checker PAT [27]. PAT is a self-contained framework for system modeling, simulation and verification. After years of development, PAT has more than 600 KLOC, thousands of test cases as well as a list of discovered bugs. In order to systematically improve PAT's quality, we propose an approach which combines code contracts with model checking techniques. The contracts serve a correctness specification and model checking is used as an effective technique to search for contract violations as well as violations of additional critical properties. Formally developing and validating a formal verification system itself is the fundamental approach to increase user's confidence in the formal tools like model checkers. Though we cannot completely verify PAT, the approach is effective and scalable.

Approach. After evaluating our options, the following approach is developed. Firstly, given that PAT is developed using C# in .NET, we make use of the code contracts project [4] and systematically express coding assumptions in the source codes. The code contracts take the form of object invariants, method precondition and post-conditions. They are used to improve testing via runtime checking as well as enable static contract verification. In our approach, the contracts serve as a partial correctness specification of PAT. All test cases are then executed to make sure that code contract violation is absent. We highlight that code contracts are not only used to capture coding assumptions on data structures or model checking algorithms but also assumptions on GUI.

Like any model checker, PAT supports many options to apply model checking in different settings, for instance, whether to use Depth-First-Search (for memory saving) or Breadth-First-Search (for shortest counterexamples); whether to apply partial order reduction; whether to apply fairness consumption while verifying liveness properties; etc. The options may conflict with each other. For instance, fairness is irrelevant when the property is safety; partial order reduction is not sound when strong fairness is assumed. The options are carefully controlled through complicated logic on user interfaces, which can be specified by code contracts.

In order to achieve another level of assurance, we develop interface models (in PAT's input language) to capture all possible scenarios in which PAT or part of it is executed. For instance, for any complicated underlying data structure, we develop an interface model which subsumes all possible ways that PAT interacts it. Models can also be developed to capture all possible ways of users interacting with PAT through GUI. The models allow us to systematically generate test cases and, better, apply model checking techniques. PAT is firstly extended to support user defined C# library, i.e., an object or method defined the C# library can be invoked as part of a model. This creates a way of linking events of the interface models with actual codes of PAT. A transition of the model is thus the result of executing certain code fragments of PAT. For instance, the event of clicking certain button in the model for user behaviors generates an actual button clicking event. *The models are then model checked using PAT* - so that all possible sequences of executing PAT codes are verified against the embedded code contracts and, in addition, properties of entire system execution history. Notice that in order to model checking the interface model for a data structure, because a data structure may often take infinite different value, empirical studies are applied to discover reasonable bounds for the values. Interface models can be developed and verified for any part of PAT, which makes this approach compositional.

We remark that the interface models may contain concurrency, which makes model checking meaningful as well as challenging. Firstly, PAT supports parallel model checking [20], which makes use of multiple CPUs to explore different parts of a system concurrently. As a result, the underlying data structure may be accessed concurrently. Secondly, PAT supports multi-threaded graphic user interface and therefore multiple simulators and model checkers can be opened simultaneously, which leads to concurrent executions. By model checking the interface models, contract violations which are the result of unlikely event sequences can be discovered systematically. For instance, bugs on GUI which are the result of a particular sequence of button clicking on multiple PAT windows have been discovered. Even though this paper is focusing on model checking the model checker itself, the approach of combining code contracts with the model checker is much more general and this approach can be applicable for checking many C# software systems.

2 PAT Background

PAT (Process Analysis Toolkit) [27] is developed as a self-contained environment to support system modeling, simulation and verification. It has user friendly model editor, animated simulator as well as fully automated model checking facility. PAT offers

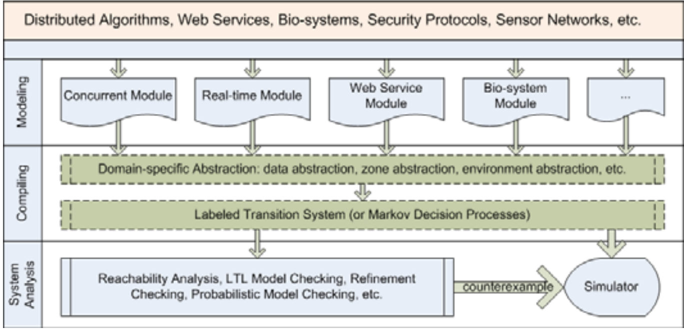


Fig. 1. PAT Architecture

a library of various model checking techniques for checking deadlock-freeness, linear temporal logics (with a variety of fairness), and refinement checking. Advanced state reduction techniques have been implemented, e.g., partial order reduction, process counter abstraction, parallel model checking. PAT has been used to model and verify a variety of systems. Previously unknown bugs have been discovered [19].

As shown in Fig. 1, PAT adopts a layered design to support analysis of different domains. For each supported domain (e.g., distributed systems, real-time systems, service oriented computing and so on), a dedicated module is created in PAT, which identifies the specialized language syntax, well-formness rules as well as formal operational semantics. The operational semantics translates a model into LTS (Labeled Transition Systems)[1] at runtime. LTS serves as a shared implicitly internal representation of the models, which can be automatically explored by the verification algorithms or used for simulation. To perform model checking on LTSs, the number of states in the LTSs must be finite. For systems with infinite states (e.g., with real time clocks or infinite number of processes), abstraction techniques are needed. Example abstraction techniques which have been realized include process counter abstraction, clock zone abstraction, environment abstraction, etc. Depending the property to verify, a proper verification algorithm is invoked. The algorithm performs on-the-fly exploration of an LTS. If a counterexample is identified during the exploration, it can be animated in the simulator. This design allows new modules to be easily plugged in and out, without recompiling the core system. This design achieves *extensible architecture* as well as *module encapsulation*.

Notice that the library of model checking algorithms as well as the GUI are shared by all modules. Their correctness are thus vital. The algorithms heavily reply on multiple highly complicated data structures, i.e., the one for system configuration; the one for compact representation of the system transition relation; the one for constraints on system clocks; etc. Not only the data structures must function correctly but also they must function efficiently. They are highly optimized, which implies that they may not be designed for rigorous system maintenance. For instance, having object orientation and recursion may not be feasible. The logic for controlling GUI is highly nontrivial as well. This is because the GUI controls different options for invoking the model checking algorithms. Different modules may employ different abstraction techniques which

[1] To be precise, it is a Markov Decision Process when probabilistic choices are involved.

conflict with certain group of model checking algorithms. For instance, over approximation of system graph conflicts with valid result for deadlock-freeness checking, i.e., an over approximated state graph is deadlock-free does not imply anything about the ordinary state graph. In such a case, if an abstraction which results in over approximation (e.g., predicate abstraction) is detected, verification result for deadlock-freeness checking must be modified properly.

PAT has been heavily tested with thousands of black-box test cases, and used daily by research and industry users. Nonetheless, bugs are still reported from time to time. Main reason for these bugs is that PAT is constantly under revision. As coding assumptions are often made in order to gain efficiency, code modification or function extension in one part of PAT often leads to coding assumption breaking in other parts, which results in new bugs. Currently, PAT has more than 600 KLOC, more than 1300 classes, 6 modules and more than 10K builds. PAT has attracted more than 800 registered users from more than 180 organizations. In summary, after years of development, PAT becomes a huge software package which requires systematically quality control.

3 Embedding Code Contracts

Code contracts [4] take the form of object invariants, preconditions, post-conditions and assertions. In a systematic way, it offers programming by design. Code contracts can be integrated into existing coding projects seamlessly, which makes them more attractive than approaches like SPEC#. Contracts are validated at run-time[2]. In the PAT project, coding assumptions are everywhere. One reason is that assumptions often make it possible to significantly simplify codes, which leads to faster model checking. Code contracts are used to capture coding assumptions on operational language semantics, model checking algorithms, underlying data structures, GUI, static model analysis functions, etc. In the following, we illustrate how code contracts are embedded systematically in PAT using two examples, one for a complex data structure which is essential to model checking real-time systems and the other for a user interface. Embedding code contracts is the first and the most essential step in our approach. They serve partially as a correctness specification of PAT.

Example 1 (Contracts for the DBM Class). Practical systems which interacts with the physical environment are often subject to quantitative timing constraints. For instance, a pacemaker must react to an abnormal heart condition within a critical time frame. Model checking real-time systems often involves manipulating constraints on multiple real-valued clocks. A timing constraint in PAT is the conjunction of multiple simple constraints. A simple constraint is of the form $c \sim d$ where c is a clock; d is a rational number; $\sim$ is a binary operator like $\geq$, $\leq$, etc. Multiple simple constraints may conflict with each other and thus make their conjunction unsatisfiable. For instance, the conjunction of $c_1 \geq 5$ and $c_2 \leq 1.5$ is unsatisfiable if c_2 is started within 3 seconds after c_1 is started (so that $c_1 - c_2 \geq 3$). During system exploration, the constraints must be stored, updated and solved efficiently. In PAT, this is achieved by techniques based on Difference Bound Matrix (DBM [9]).

[2] Contracts supports static analysis as well, which is helpful but largely irrelevant to this work.

```
public sealed class DBM {
1.      private List<List<int>> Matrix;                                    //the matrix itself
2.      private bool IsCanonicalForm = true;                               //a boolean flag
3.      private List<int> Clocks;                                          //a list of clocks
4.      //Contract Invariant Method
5.      protected void ObjectInvariant() {
6.          Contract.Invariant(!IsCanonicalForm || (IsCanonicalForm && isCF()));
7.      }
8.      //methods
9.      public void AddClock(byte cID) { ···}                              //add a new clock
10.     public void ResetClock(byte cID){ ···}                             //reset an existing clock
11.     private void GetCanonicalForm() { ···}                             //Floyd-Warshall algorithm
12.     public void AddConstraint(byte cID, OperationType op, int constant){ ···}
13.     public void Delay(){ ···}                                          //let arbitrary time pass
14.     public bool IsSatisfiable(){ ···}                                  //check satisfiability
15.     public DBM RemoveClocks(List<byte> activeClocks) {                 //remove clocks
16.         Contracts.Requires(IsCanonicalForm, "precon, failed.");
17.         Contracts.Requires(Clocks.containsAll(activeClocks), "precon. failed.");
18.         Contracts.Ensures(Matrix.Count == Clocks.Count && ···&&
19.                           IsCanonicalForm, "postcondition failed.");
20.         ···
21.     }
22.     ···
23.}
```

Fig. 2. DBM Contracts

Given n clocks $c_1, c_2, \cdots, c_n$, a DBM contains $n+1$ rows, each of which contains $n+1$ elements. Let d^i_j represent entry at i-th row and j-th column in the matrix. d^i_j represents the difference between clock c_i and c_j. A DBM represents the following constraint: $\forall\, i : 0 \,..\, n.\ \forall\, j : 0 \,..\, n.\ c_i - c_j \leq d^i_j$ where c_0 is set to be 0 all the time. The most important property of DBM is that there is a relatively efficient procedure to compute the tightest bound on each clock difference, which can be used to tell whether the constraint represented by the DBM is satisfiable or not. If the clocks are viewed as vertices in a weighted graph and the clock difference as the label on the edge connecting two clocks, the tightest clock difference is the shortest path between the respective vertices. The Floyd-Warshall algorithm [11] thus can be used to compute the tightest clock differences. A DBM which contains only tightest bounds is said to be in its *canonical form*. Given a DBM in canonical form, checking whether the constraint is satisfiable or not is as easy as checking if entry d^0_0 is positive.

DBM is implemented as a stand alone class of 1.5 KLOC. It makes use of some other simple data structures. Fig. 2 shows partially the signature of the DBM class, i.e., public methods and three relevant variables. *Matrix* is a two dimensional array storing the matrix itself; *IsCanonicalForm* is boolean flag to indicate whether the matrix is in its canonical form; *Clocks* maintains a list of active clocks. Sample code contracts are presented and underlined in Fig. 2. The invariant (line 5 to 7) states that either the DBM

is not in its canonical form or if it is, then applying an alternative method for calculating the tightest bounds (which is implemented as method $isCF()$) makes no change.

Many of the methods require that the DBM must be in its canonical form before their execution. One example is $RemoveClocks$ which removes in-active clocks and together with constraints on them. If the DBM is not in its canonical form, removing clocks might weaken the constraint. For instance, if the constraint is $c_1 \geq 3$ and $c_2 \leq 6$ and $c_1 - c_2 \leq 1$ (which implies $c_1 \geq 5$), removing c_2 results in a weaker constraint $c_1 \geq 3$. Fig. 2 shows how the pre-condition (line 16 and 17) as well as post-condition is coded as contracts (line 18 and 19). The precondition states that the DBM must be in its canonical form and the clocks to remove must be present, while the simplified postcondition states that only the clocks in $activeClocks$ remain and the DBM remains in its canonical form. Notice that the postcondition is incomplete.

Efficiency is essential to any model checker. Keeping a DBM always in its canonical form (by calling method $GetCanonicalForm$ every time the matrix is modified by $AddClock$, $AddConstraint$, etc.) is infeasible given that the Floyd-Warshall algorithm is cubic in the number of clocks. Preferably, method $GetCanonicalForm$ shall only be invoked when necessary, i.e., in method $IsSatisfiable$. The pre-condition of the methods thus must be ensured by invoking the methods in particular orders. The assumptions on orders of method invocation can be referred as *class interface contracts*. To the best of our knowledge, such contracts are not supported by the code contracts project. It is supported, recently by the SPEC explorer project for model-based testing [3]. In the next section, we show that we can build an interface model to capture *class interface contracts*, and then not only generate test cases from the model but also verify the models against meaningful properties. □

Example 2 (Contracts for GUI). There are dozens of windows that users can interact with PAT, e.g., a featured editor which has many advanced editing functions; a simulator which allows user to perform different simulation functions; and a model checking window which controls all options for applying model checking. Given that PAT supports a library of model checking algorithms as well as optimization techniques, there could be a large combinations of options to choose from when a specific model checking problem is presented. For instance, whether it should be LTL model checking or refinement checking; or whether to apply nested DFS or SCC-based search for LTL model checking; or whether the LTL model checking should be based on generating Büchi automata. There are more than 5 options for generating Büchi automata from LTL alone [12]! The options are all controlled by enabling/disabling GUI components, which as a result has a complicated and error-prone control logic. The constraints are naturally captured using object invariants associated with the GUI components.

Fig. 3 illustrates the idea with one invariant associated with one checkbox in the model checking window. The checkbox, once checked, requires the model checking algorithm to produce one shortest witness trace (often as a counterexample). If the selected property is a safety property (e.g., a reachability condition, deadlock-freeness, refinement relationship), breadth-first-search based reachability analysis or refinement checking is applied. If the selected property is a liveness property, the checkbox must be unchecked and disabled. This is because a counterexample to a liveness property must be an infinite trace (which forms a loop in finite state systems). Instead, other GUI

```
1. protected void ObjectInvariant() {
2.         Contract.Invariant(checkBox_ShortestTrace_Invariant());
3.         ...;
4. }
5. private bool checkBox_ShortestTrace_Invariant() {
6.         if (Label_SelectedAssertion != null)
8.                 if((AssertionType)cb_item.Tag == AssertionType.Liveness) {
9.                         return !CheckBox_ShortestTrace.Checked
                                       && !CheckBox_ShortestTrace.Enabled;
10.                } else { return CheckBox_ShortestTrace.Enabled; }
14.        return true;
15.}
```

Fig. 3. GUI Contracts

components like a dropdown list for fairness options, which only makes sense with liveness properties, must be enabled for selection.

In this example, the invariant states that if an assertion has been selected (from a table, which triggers update of a label *Label_SelectedAssertion* to reflect user's choice), and the selected property is a liveness property, then the checkbox must be disabled. Notice that this invariant is relaxed during the process of GUI updating. □

The contracts serve partly as a specification of the program. Once the code contracts are embedded, we firstly re-run all the test cases checking for contract violations. Our experience suggests that it is indeed possible that a test case is successful (i.e., causes no exception and produces correct output) but triggers violation of contracts during execution. One of the reasons is that the pre-condition may be irrelevant to the correctness of the output in certain cases. Detecting such cases are nevertheless useful as it helps to either find bugs or refine the specification (e.g., weakening the contract for pre-condition).

Because PAT is frequently updated, relevant code contracts must be updated as well. Coding assumptions may often change when a system evolves. Coding them explicitly as part of the system allows us to quickly detect bugs which are due to changing coding assumptions (refer to an example in Section 5).

4 Model Checking PAT

While code contracts are good at capturing intra-class coding assumptions, they are not good at capturing class level or even inter-class coding assumptions. For instance, the DBM class in PAT is designed to function correctly only under the assumption that its methods shall only be invoked in certain orders. In general, making such assumptions may not be reasonable. It is, however, common to model checkers as the assumptions may often be useful for the sake of efficiency. To test classes with implicit assumptions, all meaningful test cases must obey the assumptions. Otherwise, contract violation or even exceptions that are irrelevant to the correctness of the system may be reported. An interface model thus can capture all class-level or inter-class coding assumptions.

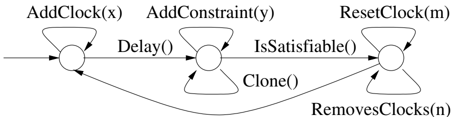


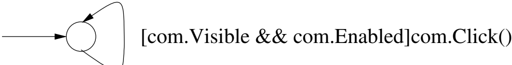
Fig. 4. DBM Model

Interface Modeling. In the following, we develop interface models to capture all valid ways of interacting with certain components of PAT or PAT as a whole. We illustrate the idea using two interface models.

Example 3 (Interface Model for DBM). Fig. 4 presents an interface model for the DBM class. The model takes the form of a finite state machine, possibly with auxiliary variables. The transitions are labeled with public methods of the DBM class. The model in Fig. 4 captures multiple class level assumptions.

For instance, method $RemoveClocks$ is always be invoked after $IsSatisfiable$ (and a method for collecting all active clocks defined in a separate class, which is omitted for the sake of space). Method $IsSatisfiable$ invokes $GetCanonicalForm$ internally and therefore it is unnecessary in method $RemoveClocks$ to apply the Floyd-Warshall algorithm or even check whether variable $IsCanonocal$ is true or not - it is always true. Similarly, this is also the case for $ResetClock$. Notice that this model is *open* to environmental inputs on the method parameters. □

Example 4 (Interface Model for GUI). The following presents an interface model for *any* user interface class. Let com be any GUI component (e.g.,any button), which users can interact with.



The model states that as long as com is visible and enabled, users can interact with it. Informally speaking, it means that no coding assumptions shall be placed upon the users and all user behaviors must be properly handled! Notice that this model can be generated automatically from the signature of any GUI class.

Users can interact with multiple user interfaces simultaneously. Therefore, the interface model for PAT contains the parallel composition of multiple interface models. Furthermore, the interface models may communicate with each other. □

Once we have the model, the task of verifying this part of the system breaks into two sub-tasks. Firstly, we need to guarantee that in the real system, interactions with the objects (of the class or classes) are permitted by the model. In this project, this is achieved with the expert knowledge of the PAT developers. In general, techniques like program slicing [29] may help, or a run-time monitor program, much like the monitor for code contracts, can be used to detect violation of class interface violation. Secondly, we need to guarantee that the system functions correctly given any behavior allowed by the model. One way of checking that is to systematically generate test cases from the

models and execute the test cases while looking for exceptions or code contracts violation [3]. Alternatively, we can model checking the models! By model checking, code contracts which are related to the entire system execution history (e.g., liveness properties) can be stated as a property and then formally verified. The models can be captured as PAT models and, hence, *we can model check the models using PAT itself!*

Supporting Runtime C#. In order to verify actual source codes, firstly we extend PAT to support dynamic loading of C# code inside a model. Our approach is to compile a C# program into a dynamic-link library (DLL) and use it during system exploration via *reflection*. By following pre-defined API, any C# class can be invoked dynamically in an interface model in PAT. Furthermore, the C# class may contain code contracts. PAT provides a contracts compilation option to automatically compile the code using contracts rewriter compiler. Any contract violation or run-time exception is presented as a runtime error to the users during model checking. In other words, ordinary PAT codes, with coding assumptions, can now become part of a PAT model with little modification.

Checking PAT. The following approach is adopted to model check PAT. Firstly, an interface model of a PAT component is written as a PAT model; the relevant C# source codes, with code contracts embedded, are compiled into an external DLL and then PAT is used to enumerate all possible behaviors of the model. If any event sequence triggers a violation of code contracts or exceptions, then a possible bug is detected. Because it is the actual code which is being executed during model checking, a discovered bug corresponds to an actual bug.

Like testing, this approach is incremental - a self-contained class can be firstly modeled and checked and then classes which rely on it can be modeled and checked. If the states searching in the model checking is viewed as simply a systematic way of generating test cases, this approach is closely related to work on model-based testing. It is, however, more than testing. For model checking, meaningful properties, which are implied from the correctness of system and cannot be validated by testing, can be verified. In general, this approach is as challenging as verification software. Many challenges like infinite data value and asynchronous thread execution must be dealt with. In the following, we use the two examples to illustrate relevant issues and our remedies.

Example 5 (Model Checking the DBM Class). A model may be open to environmental inputs. For instance, given the model for DBM class presented in Fig. 4, a clock must be supplied when an event linked to method $AddClock$ is invoked; or a simple constraint must be supplied when invoking method $AddConstraint$. In order to model check the model, it must be closed by supplying an environment. In general, there are infinite possible clocks or constraints. This problem is solved by applying empirical studies to discover reasonable bounds for the values.

Given the DBM model, we need to fix a finite set of clocks (so that x, m, n in Fig. 4 have finite values) as well as a finite set of constants for forming clock constraints (so that y has finite values). It is discovered that for most of the real-world real-time systems verified by PAT, the number of clocks are often limited to a small number (e.g., 10 or less). This is not surprising as PAT is optimized to minimize the number of clocks [28] - recall that Floyd-Warshall algorithm is cubic in the number of clocks. PAT

Table 1. Experiments: Reachability/deadlock-freeness

#DBM	#Clocks	#Constants	Reachability/deadlock		Every clock is bounded	
			#States	Time (s)	#States	Time (s)
1	1	6	7375	1.55	63641	31.4
1	1	7	12947	3.12	127362	74.1
1	1	8	21235	5.88	234254	157
1	1	9	33007	10.3	403274	310
1	2	6	473328	168	5918993	5036
1	2	7	1104560	449	15783113	15831
1	3	3	222903	64.1	2016851	1264
1	3	4	1532935	572	17659797	15431
2	1	3	511225	172	?	?

only introduces a clock whenever necessary; a clock is shared as much as possible and a clock is removed as soon as possible. Furthermore, there are only a relatively small number of constants for forming clock constraints in most cases. Once we fix the clocks and constants, we have a finite model which is subject to model checking.

Because PAT supports parallel model checking, any data structure used by the model checking algorithms may be accessed in parallel by multiple threads. This may create problems like race condition. Even if the objects are not shared by the threads, static variables or references which are accessed by the objects directly or indirectly are always shared. Static variables allow quick access of information. They, however, must been properly locked and unlocked if accessed concurrently. We thus extend the model to capture concurrent accessing of data objects, e.g., the DBM, by composing multiple copies of the interface models in parallel. Furthermore, the exact locking mechanism used in PAT has been modeled and reflected in the model as well. Different properties can be formulated and model checked against the interface models. In general, a property is necessary condition of the correctness of the PAT. By verifying that the properties are true, we gain confidence in the system's correctness.

In the following, we illustrate three important properties of DBM and use PAT to model check the respective model. Firstly, an unsatisfiable reachability condition is model checked, which triggers exploration of the complete state space. This allows us to verify that the embedded code contracts are satisfied in all system configurations. In addition, we verify that the DBM model is deadlock-free. This is particularly interesting with more than one DBM objects - it should not be that two DBM objects are both waiting to access a shared object. The experiment results are summarized in column "Reachability/deadlock" of Table 1. The results are obtained on a PC with Intel Xeon 4-Core CPU*2, 32 GB memory, with fixed number of DBM objects, number of clocks and number of constants. This property is verified using an on-the-fly reachability analysis algorithm in PAT. After correcting serval bugs, the result is eventually all false, as expected. Secondly, a property stating that every clock is bounded is verified to confirm a fundamental theorem that every clock can take only finite different ranges (which is known as zones [27]). The theorem is one of two necessary conditions to guarantee that model checking of real-time systems in PAT is always terminating. It is an important property of DBM which can be proved from the formal semantics of the real-time

Table 2. Experiment C: Every clock always eventually expires

#DBM	#Clocks	#Constants	#States	Result	Fairness	Time (s)
1	2	3	58234	false	no fair	26.7
1	3	3	1445821	false	no fair	2229
1	1	3	1676	false	weak fair	0.575
1	2	3	12372	false	weak fair	26.7
1	3	3	1445821	false	weak fair	2223
1	1	3	1676	false	strong fair	0.643
1	2	3	58234	false	strong fair	29.7
1	3	3	1445821	false	strong fair	2348
1	1	3	5020	true	global fair	2.19
1	2	3	148860	true	global fair	353
1	3	3	3462673	true	global fair	244733

system modeling language supported in PAT [27]. The experiment results are summarized in column "Every clock is bounded" of Table 1, where a question mark means out of memory. Lastly, Table 2 summarizes our experiments on verifying the other necessary condition, which too can be proved from the semantics model. Intuitively, the property states that every clock *always eventually* expires. We highlight this is a liveness property which cannot be validated by testing. It is captured as an LTL formula and verified using the automata-based verification algorithm in PAT. Furthermore, this property is valid only under certain fairness constraint [27], which intuitively says that there are only finitely many system actions within one time unit. Notice that verifying LTL properties with strong fairness is a unique feature of PAT.

If a counterexample is generated, a unit test case is generated from the counterexample straightforwardly (since the counterexample is a sequence of method calls with input values) so as to locate the bug. As expected, the experiments show that model checking suffers from the state space explosion problem, e.g., parallel execution of multiple DBM objects results in huge number of system configurations. Furthermore, because of the bounds, only part of all possible behaviors are examined. Nonetheless, large number of system executions are examined systematically (with blind cases) against complex properties, which greatly increase our confidence in system correctness. Furthermore, because of the empirical studies, we focus on common cases and therefore reduce the probability of producing wrong results in common practice of PAT. □

Example 6 (Model Checking the GUI). In order to model check PAT as a whole, we build a model to capture all possible ways of users interacting with PAT. The model is composed of each and every user interface model in the order which users can interact with them. Each event in the model is linked to an actual GUI event which triggers execution of PAT. Model checking is then applied to enumerate all event sequences and validate them against the embedded contracts, including those which are hard to create in practice. For instance, the following scenario is found to lead to contract violation. A user firstly clicks one button to initiate a thread for state graph generation in one simulation window, which triggers an attempt to lock shared (static) variables. The user then clicks the same button in another simulation window. If two models happen to share

common process definitions and two models are not identical, then the first simulation window may behave wrongly because an indirectly accessed variable is changed when the user clicks the second button.

The GUI model essentially creates a “robot” which controls PAT, which allows users to manipulate PAT arbitrarily. We can then easily create and verify complicated properties which may require multiple execution of multiple model checking algorithms. For instance, one interesting theory is that if a model satisfies a property under weak fairness, then it must satisfy the property under stronger fairness constraints like strong fairness or strong global fairness [27]; if a counterexample is produced when a property is verified under strong fairness, then a counterexample must be produced when the property is verified under weak fairness. In order to check this is indeed true, a simple GUI model is created to load one built-in case study at a time, perform model checking under weak fairness, perform model checking under strong fairness and then compare whether the results are as expected. Many theorems implied from the correctness of PAT can be checked in this way.

One difficulty in checking GUI models is that different GUI components may run as different threads. Multiple threads may execute simultaneously. For instance, one occurrence of the event for clicking the verification button triggers the creation of a thread for model checking. Before the model checking completes, the user can click another button to trigger another thread. The threads are scheduled by the system scheduler. Different executing of the same event sequence of the model may result in different system configuration due to different run-time scheduling. This is a known problem to GUI testing or testing of concurrent programs in general. A common remedy is to run a test case sufficient number of times (or with inserted random thread sleep) so as to exhaust all possible scheduling. Notice that because the model checking of the GUI model largely depends on the size of the input model, we omit the statistics.

5 Discussion

In this section, we discuss the limitation of our approach and related works.

Limitations. Firstly, *we cannot completely verify PAT*. Compromises have been made in order to deliver a useful technique handling PAT. For instance, the code contracts only capture part of the correctness specification; we can only verify part of the behaviors of an interface model, etc. Secondly, when model checking a component of the system, assumptions on the rest of the system are often necessary. For instance, the input to method $RemoveClocks$ is assumed to be a set of clocks, which assumes that the method for obtaining the clocks removes any redundancy. Systematically verifying the assumptions are highly non-trivial. This is a known problem which has been discussed in [6]. Lastly, generating properties to be model checked needs expert knowledge on the underlying theories of the system.

The infamous state-space explosion problem still exists. For instance, given a DBM object with N clocks, there are 2^N different inputs to method $RemoveClocks$. This problem is known to be best solved by methods like data abstraction [2], which currently remains as one of our future work. But still, model checking remains useful even if only

part of the system behaviors are explored. It explores all possible behaviors of a model, including corner cases which are unlikely for real-world applications. Compared to model-based testing, the additional properties are often useful in gaining confidence of the implementation or the theorems which lead to the properties. Our experience is that given all model checking algorithms have multiple theorems behind (e.g., for soundness, completeness and termination), many properties can be deduced naturally.

Related Work. To our best of knowledge, this work is the first attempt on using advanced system analysis techniques (e.g., code contracts and model checking) to systematically validate a model checker. Our approach is related to numerous work on software verification, testing and debugging.

Our approach relies on programming by contracts. In particular, the code contracts project essentially makes it possible [4]. There are other methods for embedding specification into programs. Noticeable examples are SPEC# for C# and JML for Java. We remark that as long as there is a way of run-time checking the specification, any method would work in our approach. We choose code project simply because it is the option which requires minimum modification to our programs.

Our approach can be categorized as combining programming by contracts with model checking. It is closed related to work on combining programming by contracts with model-based testing. The tool SPEC explorer supports model-based testing in addition to contract embedding [3]. Similar to our approach, SPEC explorer uses a model checker so it can enumerate all possible sequences of method invocations that do not violate precondition or invariant of the system's contracts. Furthermore, users are allowed to specify a set of testing properties, which plays a similar role as our interface model. Our approach is different from model-based testing [16] in the following ways. Firstly, we perform model checking instead of model-based testing. The difference is that besides exception-freeness and no violation of contracts, additional properties regarding to interface models can be verified. The properties may not be validated by testing. The properties are often implied from the underlying theorems. By model checking them, not only we gain confidence on PAT but also the theorems. In addition, our work targets at validating a model checker. We use the model checker to check the interface models as a way to verify itself.

This work is related to work on software verification [1,26]. Apart from very constrained scenarios (e.g., verification of device drivers), the software verification tools are not widely used in general software development process. The main reason is that they do not scale. For instance, the SLAM project is based on data abstraction, which is a complicated and computationally expensive process. In contrast, our approach is scalable. Even partial code contracts are useful. Like testing, our approach is compositional in the sense that each time part of PAT can be checked and then their composition. Furthermore, it is compatible with rapidly evolving programs. Once a few relevant contracts need to be updated every time system evolves. As a reasonable price to pay, we can not verify PAT altogether. Nonetheless, our approach helps to significantly improve stability and reliability.

The work is related to work on specification and verification of object interfaces [15,5]. The main difference is that we combine code contracts. This work is related to work on combining testing with model checking [13,18,23]. In [18], a tool named

UnitCheck is present which allows creation, execution and evaluation of testing cases using the Java Pathfinder model checker. Unlike [18] where models are automatically extracted from programs, interface models are provided by PAT developers as an additional code contracts. In addition, this work is remotely related to work on testing of concurrent software [24,25], GUI testing and testing of evolving programs.

6 Conclusion

Model checkers are specialized software whose correctness are vital. In this work, we propose the combination of code contracts and model checking as a way to systematically improve their quality. The combination is effective in combating the complexity of software. Three levels of system specification are handled using the proposed approach. First is the specification of a method or a single class, captured in the form of pre/post-condition or class invariants and validated using run-time checking facility from the code contract project. Second is safety properties of a single class or a group of coupled classes, captured using interface models. It can be verified by model-based testing or reachability analysis. Lastly, specification of the entire system execution are verified against the models by model checking techniques.

We experiment the approach in checking the correctness of the PAT model checker. Through the experiments, we discovered multiple bugs, and gained more confidence of PAT. One of our future is to combine data abstraction (e.g., predicate abstraction) in model checking the interface models.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)
2. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. STTT 5(1), 49–58 (2003)
3. Barnett, M., DeLine, R., Fähndrich, M., Jacobs, B., Leino, K.R.M., Schulte, W., Venter, H.: The SPEC# Programming System: Challenges and Directions. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 144–152. Springer, Heidelberg (2008)
4. Barnett, M., Fähndrich, M., de Halleux, P., Logozzo, F., Tillmann, N.: Exploiting the Synergy between Automated-test-generation and Programming-by-contract. In: ICSE Companion 2009, pp. 401–402. IEEE, Los Alamitos (2009)
5. Bierhoff, K., Aldrich, J.: Lightweight Object Specification with Yypestates. In: ESEC/SIGSOFT FSE 2005, pp. 217–226. ACM, New York (2005)
6. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Jurdzinski, M., Mang, F.Y.C.: Interface Compatibility Checking for Software Modules. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 428–441. Springer, Heidelberg (2002)
7. 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)
8. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1981)

9. Dill, D.L.: Timing Assumptions and Verification of Finite-State Concurrent Systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
10. Emerson, E.A., Wahl, T.: Dynamic Symmetry Reduction. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 382–396. Springer, Heidelberg (2005)
11. Floyd, R.W.: Algorithm 97: Shortest Path. Commun. ACM 5(6), 345 (1962)
12. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
13. Gunter, E.L., Peled, D.: Model checking, Testing and Verification Working Together. Formal Asp. Comput. 17(2), 201–221 (2005)
14. Holzmann, G.J.: The Model Checker SPIN. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
15. Hughes, G., Bultan, T.: Interface Grammars for Modular Software Model Checking. In: ISSTA 2007, pp. 39–49. ACM, New York (2007)
16. Jacky, J., Veanes, M., Campbell, C., Schulte, W.: Model-Based Software Testing and Analysis with C#. Cambridge University Press, Cambridge (2008)
17. Kaivola, R., Ghughal, R., Narasimhan, N., Telfer, A., Whittemore, J., Pandav, S., Slobodová, A., Taylor, C., Frolov, V., Reeber, E., Naik, A.: Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009)
18. Kebrt, M., Sery, O.: UnitCheck: Unit Testing and Model Checking Combined. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 97–103. Springer, Heidelberg (2009)
19. Liu, Y., Pang, J., Sun, J., Zhao, J.: Verification of Population Ring Protocols in PAT. In: TASE 2009, pp. 81–89. IEEE Computer Society, Los Alamitos (2009)
20. Liu, Y., Sun, J., Dong, J.S.: Scalable Multi-core Model Checking Fairness Enhanced Systems. In: ICFEM 2009. LNCS, vol. 5885, pp. 426–445. Springer, Heidelberg (2009)
21. Mühlberg, J.T., Lüttgen, G.: Blasting Linux Code. In: Brim, L., Haverkort, B.R., Leucker, M., van de Pol, J. (eds.) FMICS 2006 and PDMC 2006. LNCS, vol. 4346, pp. 211–226. Springer, Heidelberg (2007)
22. Peled, D.: Combining Partial Order Reductions with On-the-fly Model-Checking. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 377–390. Springer, Heidelberg (1994)
23. Peled, D.: Model Checking and Testing Combined. In: Baeten, J.C.M., Lenstra, J.K., Parrow, J., Woeginger, G.J. (eds.) ICALP 2003. LNCS, vol. 2719, pp. 47–63. Springer, Heidelberg (2003)
24. Sen, K., Marinov, D., Agha, G.: CUTE: a Concolic Unit Testing Engine for C. In: ESEC/SIGSOFT FSE 2005, pp. 263–272. ACM, New York (2005)
25. Sherman, E., Dwyer, M.B., Elbaum, S.G.: Saturation-based Testing of Concurrent Programs. In: ESEC/SIGSOFT FSE 2009, pp. 53–62. ACM, New York (2009)
26. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining Symbolic Execution with Model Checking to Verify Parallel Numerical Programs. ACM Trans. Softw. Eng. Methodol. 17(2) (2008)
27. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: Towards Flexible Verification under Fairness. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
28. Sun, J., Liu, Y., Dong, J.S., Zhang, X.: Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In: ICFEM 2009. LNCS, vol. 5885, pp. 581–600. Springer, Heidelberg (2009)
29. Weiser, M.: Program Slicing. In: ICSE, pp. 439–449 (1981)

On Symmetries and Spotlights – Verifying Parameterised Systems

Nils Timm and Heike Wehrheim

Department of Computer Science, University of Paderborn
D-33098 Paderborn, Germany
{timm84,wehrheim}@uni-paderborn.de

Abstract. Parameterised model checking is concerned with verifying properties of arbitrary numbers of homogeneous processes composed in parallel. The problem is known to be undecidable in general. Nevertheless, a number of approaches have developed verification techniques for certain classes of parameterised systems. Here, we present an approach combining *symmetry* arguments with *spotlight* abstractions. The technique determines (the size of) a particular *instantiation* of the parameterised system from the given temporal logic formula, and feds this into an abstracting model checker. The *degree* of abstraction with respect to processes occurring during model checking determines whether the obtained result is also valid for all other instantiations. This enables us to prove *safety* as well as *liveness* properties (specified in full CTL) of parameterised systems on very small instantiations.

Keywords: symmetry reduction, spotlight abstraction, parameterised verification, model checking.

1 Introduction

Parameterised systems consist of an arbitrary number of homogeneous processes usually composed in parallel. The objective in verifying parameterised systems is to show certain correctness properties regardless of the number of processes involved. Examples can be found in all sorts of distributed algorithms, like mutual exclusion, leader election or cache coherence. Verifying parameterised systems is undecidable in general [3]. Model checking can of course prove properties for particular instantiations by fixing the number of processes, however, this is limited to reasonably sized numbers.

Nevertheless, a lot of approaches have been developed which verify particular classes of parameterised systems, or which devise methods that are sound but not complete. These include techniques which apply *regular model checking* [2], *induction* [15] or decision procedures for *second order logics* [4] to the verification of parameterised systems. Regular model checking represents sets of states by regular expressions and performs a reachability analysis by means of transducers. This technique is quite costly as it involves a number of automata theoretic constructions. In [1] a more efficient method has been proposed which however

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 534–548, 2010.
© Springer-Verlag Berlin Heidelberg 2010

can only treat safety properties. The approach in [4] models parameterised systems in the logic WS1S and computes an abstraction of it on which properties can be shown. This technique requires to manually define abstraction relations. The invisible invariants technique of [15] computes inductive invariants on small instantiations by model checking and uses theorem proving for showing these to be inductive on the parameterised system as well. The method of [7] also uses model checking on small instances, where the number of instances (the *cutoff*) is computed from the description of the parameterised system, given this satisfies a particular format.

In this paper, we base our method on symmetry arguments. Symmetry and symmetry reductions [8,10] have long been proposed to reduce the state space in model checking. The general idea is to consider symmetric states, which only differ in permutations of values, as equivalent. Instead of constructing the whole state space, only equivalence classes (orbits) are built. These symmetries may refer to data values as well as process names. Parametric systems are inherently symmetric: usually either all processes are similar, or they can at least be divided into (a finite number of) classes of similar processes. The symmetry idea is applied in [7] to compute cutoffs, and in a sense also in [16], which counts the number of (symmetric) processes being in particular states.

Similar to [7] we will compute the size of an instance of the parameterised system. However, this size is not determined from the system description but straightforwardly from the temporal logic formula used to specify the correctness property. The size is simply 1 plus the number of different process *variables* occurring in the formula. If we for instance want to show a mutual exclusion property over a parameterised system, specified as

$$\forall\, i, j, i \neq j : AG\neg(i@crit \wedge j@crit)$$

(no processes i and j can ever be at their critical sections at the same time), the size is three (one process in addition to those mentioned). The parameterised system is instantiated with this number and the instantiation is fed into the model checker 3Spot [17]. 3Spot is our three-valued model checker [17] which employs both predicate abstraction and *spotlight abstraction* [18]. The third value "don't know" is used to represent unknowns which may arise due to the abstraction. Since we use a three-valued logic both true and false results can be transferred to the unabstracted system, only an unknown result necessitates abstraction refinement.

It is however the principle of spotlight abstraction which helps us towards our goal of proving (or disproving) the property not only for the particular instance but for the parameterised system as a whole. Spotlight abstractions completely abstract away the processes whose behaviour is irrelevant for the property to be checked. If the additional process in the instantiation is not drawn into the spotlight during the model checking run (i.e. completely abstracted away), the obtained result is valid for the parameterised system as well. Thus, it is the *degree of abstraction* occurring during the model checking run which tells us whether we can transfer our result to any number of processes. Since we employ

a three-valued model checker this holds for "true" and "false" results. Only in case that the additional process is moved into the spotlight, the result tells us nothing about the parameterised system.

The method currently works for completely symmetric systems (all processes the same, statements do not depend on process identifiers) as well as systems in which the processes can be divided into classes of similar programs. It allows for communication via shared variables. We exemplify the technique on a simple mutual exclusion and a readers/writers algorithm. The technique can be classified as being completely automatic, usually carrying out checks on very small instances (since the properties usually refer to a very limited number of different processes, most often just two) and thus being fast, and being sound but not complete.

2 Definitions

We look at systems of the following form (called *fully symmetric systems*):

$$\texttt{global } u_1 : \texttt{type}, \ldots, u_l : \texttt{type where } \varphi_{ginit}$$

$$\|_{i=1}^{n} P_i :: \begin{bmatrix} \texttt{local } v_1 : \texttt{type}, \ldots, v_h : \texttt{type where } \varphi_{linit} \\ \textit{some program text} \end{bmatrix}$$

where $u_1, u_2 \ldots$ are *global* variables from some set V_g, $v_1, v_2 \ldots$ are local variables from a set V_l and P_1 to P_n are identical processes. For both local and global variables an initialisation is given in terms of a formula φ ($\varphi_{linit}, \varphi_{ginit}$, respectively). We assume this to give us a unique initial value for all variables (initialisation predicates are deterministic). We furthermore implicitly assume the set of local variables to contain a dedicated variable *pc*, a program counter. The numbers 1 to n act as *process identifiers*, from a set $PID = [1..n]$. Within a process P_i, its process id cannot be referred to, and the local state of P_i is not accessible by P_j, $i \neq j$. The processes are completely symmetric, i.e. each process P_i executes exactly the same code. For convenience we sometimes just write $P = \|_{i=1}^{n} P_i$.

The following example of a mutual exclusion algorithm by means of a semaphore serves for illustrating our technique for fully symmetric systems (written in a language similar to SPL [13]):

$$\texttt{global } y : \texttt{ sem where } y = 1$$

$$\|_{i=1}^{n} P_i :: \begin{bmatrix} \texttt{loop forever do} \\ \begin{bmatrix} 0: & \texttt{Non-Critical} \\ 1: & \texttt{request } y; \\ 2: & \texttt{Critical} \\ 3: & \texttt{release } y; \end{bmatrix} \end{bmatrix}$$

A *state* of such a system consists of a valuation of the global variables V_g and - for every process - valuations of the local variables, with values taken from some

domain D. We define $V = V_g \cup (V_l \times PID)$ to be the overall set of variables and hence a state is a mapping $s : V \to D$. We refrain from explicitly defining types and type-preserving assignments. We assume to have - amongst others - a domain for locations called Loc, and the variable pc is assigned to values of Loc only. The valuation of a global variable $v \in V_g$ in a state s is denoted by $s(v)$, the valuation of a local variable $v \in V_l$ of a process P_i in s is denoted by $s(v, i)$. The set $V_i = V_g \cup V_l$ is the set of variables of a single process P_i, and we write $s[i]$ to describe the local view of P_i on a state s: for $v \in V_g$, $s[i](v) = s(v)$, and for $v \in V_l$, $s[i](v) = s(v, i)$.

Transitions in such systems are caused by some local process P_i executing its next statement (if enabled). The transitions of process P_i are described by a predicate R_i on primed and unprimed global and local variables. The predicate R_i is derived from the program text, and consequently in our setting of fully symmetric systems predicates R_i are the same for all $i \in PID$. The predicate R_i for the mutual exclusion example is

$$\begin{array}{l}(pc = 0 \wedge pc' = 1 \wedge y' = y)\\ \vee (pc = 1 \wedge y = 1 \wedge pc' = 2 \wedge y' = 0)\\ \vee (pc = 2 \wedge pc' = 3 \wedge y' = y)\\ \vee (pc = 3 \wedge y = 0 \wedge y' = 1 \wedge pc' = 0)\end{array}$$

We write $R_i(s[i], s'[i])$ to describe the case when the predicate R_i is true for the local views: $(s[i], s'[i]) \models R_i$.

As a computational model for our systems we use Kripke structures.

Definition 1. *A* Kripke structure *over a set of atomic propositions AP is a 4-tuple $K = (S, s_0, R, L)$ where*

- *S is a set of states,*
- *$s_0 \in S$ is the initial state,*
- *$R : S \times S \to \{true, false\}$ is a transition function,*
- *$L : S \times AP \to \{true, false\}$ is a function labelling states with atomic propositions.*

A path τ of a Kripke structure K is an infinite sequence of states $s_0 s_1 s_2 ...$ with $R(s_i, s_{i+1}) = true$; τ_i denotes the i-th state of τ and T_s denotes the set of all paths starting in $s \in S$.

As atomic propositions we use the following: $(v = d)$ for global variables $v \in V_g$ and data values $d \in D$, $(i@l)$ for a process id i and a location l, and $(v@i = d)$ for a local variable $v \in V_l$, a process id i and a data value d. For a symmetric system $P =||_{i=1}^{n} P_i$, we define its Kripke structure $K = (S, s_0, R, L)$ as follows:

- States: $S := V \to D$ (the set of type-preserving valuations to variables),
- Initial state: $s_0 := s \in S$ with $s \restriction V_g \models \varphi_{ginit} \wedge \forall\, i \in PID : s[i] \models \varphi_{linit}$ (due to the initialisation predicates being deterministic, this gives us the same initial values for the local variables in all processes),

– Transition relation:

$$R(s,s') := \exists\, i \in [1..n] : R_i(s[i], s'[i]) \\ \wedge\ \forall\, j \neq i, \forall\, v \in V_l : s[j](v) = s'[j](v),$$

– Labelling function:

$$L(s,p) := \begin{cases} s(v) = d & \text{for } \ p \mathrel{\widehat{=}} (v = d) \\ s(pc,i) = l & \text{for } \ p \mathrel{\widehat{=}} (i@l) \\ s(v,i) = d & \text{for } \ p \mathrel{\widehat{=}} (v@i = d). \end{cases}$$

For specifying properties of Kripke structures we use the computational tree logic (CTL). In the next section we will see that CTL is the base logic only, the properties we actually like to show about parameterised systems are a little more complicated since we wish to quantify over processes.

Definition 2. *Let AP be a set of atomic propositions and $p \in AP$. The* syntax of CTL *is given by*

$$\psi ::= p \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid EX\psi \mid AX\psi \mid EF\psi \mid \\ AF\psi \mid EG\psi \mid AG\psi \mid E[\psi U \psi] \mid A[\psi U \psi].$$

The validity of a CTL formula ψ on a Kripke structure K is denoted by $K, s_0 \models \psi$.

Definition 3. *Let $K = (S, s_0, R, L)$ be a Kripke structure over AP, $p \in AP$ and $\psi \in CTL$. Then the evaluation of ψ in a state s of K, $K, s \models \psi$, is inductively defined as follows*

$$\begin{array}{ll} K,s \models p & := L(s,p) \\ K,s \models \neg\psi & := \neg(K,s \models \psi) \\ K,s \models \psi \vee \psi' & := K,s \models \psi \ \vee \ K,s \models \psi' \\ K,s \models EX\psi & := \bigvee_{s' \in S} R(s,s') \wedge K,s' \models \psi \\ K,s \models EG\psi & := \bigvee_{\tau \in T_s} \bigwedge_{i \in \mathbb{N}} (K,\tau_i \models \psi) \\ K,s \models E[\psi U \psi'] & := \bigvee_{\tau \in T_s} \bigvee_{i \in \mathbb{N}} ((K,\tau_i \models \psi') \wedge \bigwedge_{0 \leq j < i} (K,\tau_j \models \psi)) \end{array}$$

(The remaining CTL operators can be derived by the usual dualities.)

3 Symmetries

Here, we look at Kripke structures representing parameterised systems $P = ||_{i=1}^{n} P_i$. We like to show properties of such systems for all the processes in it, thus our property formulae take the following form:

$$\forall\, i_1, \ldots, i_d, (i_l \neq i_j)_{1 \leq l,j \leq d, l \neq j} : \psi(i_1, \ldots, i_d)$$

where i_1 to i_d are variables for process identifiers and $\psi \in$ CTL. The number d of process variables appearing in a formula F is called the *process diameter*

of F. The following two formulae express properties we would like to show for our mutual exclusion algorithm: $F_1 : \forall\, i, j, i \neq j : AG\neg(i@2 \wedge j@2)$ (a safety property about mutual exclusion, process diameter 2) and $F_2 : \forall\, i, j, i \neq j : AG((i@2 \wedge j@1) \Rightarrow AF(j@2))$ (a liveness property about starvation freedom, process diameter 2).

The ultimate goal is to show such properties $F(i_1, \ldots, i_d)$ for any number of processes running in parallel, i.e. show that $\forall\, N > d : K, s_0 \models F(i_1, \ldots, i_d)$ where K is the Kripke structure representing $||_{i=1}^{N}\, P_i$.

For the verification we want to exploit the symmetry in such systems and therefore use a technique inspired by symmetry reductions [8]. The symmetries concern process ids only, i.e. we permute processes, but no data variables.

Definition 4. *A* process permutation *is a bijective function* $\pi : PID \rightarrow PID$.

Process permutations can be lifted to states: for a state s we define $\pi(s)$ as follows: $\pi(s)(v) = s(v)$ in case of global variables $v \in V_g$, and $\pi(s)(v, i)$ is $s(v, \pi(i))$ in case of local variables including the program counters. Process permutations can also be applied to atomic propositions: $\pi(v = d) = (v = d), \pi(i@l) = (\pi(i)@l)$ and $\pi(v@i = d) = (v@\pi(i) = d)$.

For using process permutations for verification, they should in some sense preserve the semantics of systems:

Definition 5. *A process permutation π is a* symmetry *for a Kripke structure $K = (S, s_0, R, L)$ if the following conditions are met*

1. $R(s, s') \Leftrightarrow R(\pi(s), \pi(s'))$,
2. *for all atomic propositions $p \in AP$:* $L(s, p) \Leftrightarrow L(\pi(s), \pi(p))$,
3. $\pi(s_0) = s_0$.

Note that in contrary to permutations used in classical symmetry reduction, we apply the permutation on the atomic propositions as well, i.e. we do not require $L(s, p) \Leftrightarrow L(\pi(s), p)$.

Proposition 1. *On fully symmetric systems all process permutations are symmetries.*

Proof. First, we look at the transition relation. We only show the '⇒'-direction. The '⇐'-direction is proved analogously.

$$
\begin{aligned}
R(s, s') \Rightarrow\quad & \exists\, i \in [1..n] : R_i(s[i], s'[i]) \\
& \wedge\ \forall\, j \neq i, \forall\, v \in V_l : s[j](v) = s'[j](v) \\
\Rightarrow\quad & \exists\, h = \pi(i) : R_{\pi(i)}(\pi(s[i]), \pi(s'[i])) \\
& \wedge\ \forall\, j \neq h, \forall\, v \in V_l : \pi(s[j])(v) = \pi(s'[j])(v) \\
\Rightarrow\ & R(\pi(s), \pi(s'))
\end{aligned}
$$

Second, the labelling function. Here, we have to distinguish three cases:

1. $p \mathrel{\widehat{=}} (v = d)$:
   $L(s,(v = d))$
   $\Leftrightarrow s(v) = d$
   $\Leftrightarrow \pi(s)(v) = d$
   $\Leftrightarrow L(\pi(s),(v = d))$
   $\Leftrightarrow L(\pi(s), \pi(v = d))$
2. $p \mathrel{\widehat{=}} (i@l)$:
   $L(s,(i@l))$
   $\Leftrightarrow s(pc, i) = l$
   $\Leftrightarrow \pi(s)(pc, \pi(i)) = l$
   $\Leftrightarrow L(\pi(s), \pi(i)@l)$
   $\Leftrightarrow L(\pi(s), \pi(i@l))$
3. $p \mathrel{\widehat{=}} (v@i = d)$:
   $L(s,(v@i = d))$
   $\Leftrightarrow s(v, i) = d$
   $\Leftrightarrow \pi(s)(v, \pi(i)) = d$
   $\Leftrightarrow L(\pi(s), \pi(v@i = d))$ □

If a process permutation is a symmetry, then we can also permute paths in the Kripke structure, thereby again getting valid paths.

Proposition 2. *Let π be a symmetry for a Kripke structure $K = (S, s_0, R, L)$ and $\tau = s_0 s_1 s_2 \ldots$ a path in K. Then $\pi(\tau) = \pi(s_0)\pi(s_1)\pi(s_2)\ldots$ is a path in K as well with $L(s_i, p) = L(\pi(s_i), \pi(p))$ for all atomic propositions $p \in AP$ and $i \in \mathbb{N}$.*

This is a key property for our symmetry argument since it lets us prove properties about certain subsets of processes which will then also hold for other subsets gained by permutation. In general our approach to verifying properties about symmetric systems works as follows. For a formula $\forall\, i_1, \ldots, i_d : \psi(i_1, \ldots, i_d)$ we first of all only consider the temporal logic part, i.e. $\psi(i_1, \ldots, i_d)$. For determining its validity for a symmetric system, we instantiate the process variables i_j with concrete values $p_j \in PID$ (pairwise different) and look at the formula $\psi(p_1, \ldots, p_d)$. Our first theorem states that the application of a permutation does not change the validity of such properties.

Theorem 1. *Let $||_{i=1}^{n} P_i$ be a fully symmetric system, $K = (S, s_0, R, L)$ the corresponding Kripke structure over AP and π a process permutation which is a symmetry for K. Moreover, let ψ be a CTL formula and $s \in S$. Then*

$$K, s \models \psi(p_1, \ldots, p_d) \;\Leftrightarrow\; K, \pi(s) \models \psi(\pi(p_1), \ldots, \pi(p_d)) \;.$$

Proof. Induction on the structure of ψ. The argumentation is based on Definition 5 and Proposition 2. For short we write ψ for $\psi(p_1, \ldots, p_k)$ and $\pi(\psi)$ for $\psi(\pi(p_1), \ldots, \pi(p_k))$.

– $\psi = p,\ p \in AP$:
$K, s \models p$
$\Leftrightarrow\ L(s, p)$
$\Leftrightarrow\ L(\pi(s), \pi(p))$
$\Leftrightarrow\ K, \pi(s) \models \pi(p)$

– $\psi = \neg\psi_1$:
$K, s \models \psi$
$\Leftrightarrow\ K, s \not\models \psi_1$
$\Leftrightarrow\ K, \pi(s) \not\models \pi(\psi_1)$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = \psi_1 \vee \psi_2$:
$K, s \models \psi$
$\Leftrightarrow\ K, s \models \psi_1 \vee K, s \models \psi_2$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi_1) \vee K, \pi(s) \models \pi(\psi_2)$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = EX\psi_1$:
$K, s \models \psi$
$\Leftrightarrow\ \bigvee_{s' \in S} R(s, s') \wedge K, s' \models \psi_1$
$\Leftrightarrow\ \bigvee_{\pi(s') \in S} R(\pi(s), \pi(s')) \wedge K, \pi(s') \models \pi(\psi_1)$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = EG\psi_1$:
$K, s \models \psi$
$\Leftrightarrow\ \bigvee_{\tau \in T_s} \bigwedge_{i \in \mathbb{N}} (K, \tau_i \models \psi_1)$
$\Leftrightarrow\ \bigvee_{\pi(\tau) \in T_{\pi(s)}} \bigwedge_{i \in \mathbb{N}} (K, \pi(\tau_i) \models \pi(\psi_1))$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$

– $\psi = E[\psi_1 U \psi_2]$:
$K, s \models \psi$
$\Leftrightarrow\ \bigvee_{\tau \in T_s} \bigvee_{i \in \mathbb{N}} ((K, \tau_i \models \psi_2) \wedge \bigwedge_{0 \leq j < i} (K, \tau_j \models \psi_1))$
$\Leftrightarrow\ \bigvee_{\pi(\tau) \in T_{\pi(s)}} \bigvee_{i \in \mathbb{N}} ((K, \pi(\tau_i) \models \pi(\psi_2)) \wedge \bigwedge_{0 \leq j < i} (K, \pi(\tau_j) \models \pi(\psi_1)))$
$\Leftrightarrow\ K, \pi(s) \models \pi(\psi)$ □

This is all we need with respect to symmetries: for symmetric systems a property is true if and only if its permutation is true. Up to here, we however have no means of dealing with the $\forall N > d$ in our formulae. For parameterisation we next combine this technique with spotlight abstractions.

4 Spotlights

By having instantiated our formula with concrete process ids, we have obtained a *local* property, referring to only some of the processes in PID. Instantiations of F_1 and F_2 could be $AG\neg(1@2 \wedge 2@2)$ and $AG((1@2 \wedge 2@1) \Rightarrow AF(2@2))$, respectively. Local properties of parallel processes can be checked using spotlight

abstractions and the tool 3Spot. 3Spot is a verification tool based on the concept of predicate abstraction and abstraction refinement [17], which - however - does not only apply predicate abstraction to the code of processes in a parallel composition but also abstracts away complete processes. Those processes that are referred to in the property to be checked are taken into the *spotlight* whereas all others are kept in the *shade*. On the processes in the spotlight we apply ordinary predicate abstraction. The processes in the shade are automatically abstracted into one approximative process $P_\perp$. This process only coarsely reflects the behaviour of the shade processes. In particular, $P_\perp$ neglects the original control flow of the processes in the shade. Instead it approximates operations on global variables occurring in shade processes by continuously modifying predicates over those variables. Due to the approximative character of $P_\perp$ and the inherent loss of information about the shade processes, predicates might be set to *unknown*. "Unknown" is in fact a valid value as we operate with three-valued logics (Kleene logic [9]), where predicates can be true, false or unknown.

Spotlight abstraction now works as follows:

1. We start with a spotlight which only contains those processes the property formula speaks about. For the verification, we construct a predicate abstraction of these processes and combine this in parallel with the approximative process representing all processes in the shade.
2. On this abstraction the formula is checked. If the check returns true or false, we are done. The result also holds for the non-abstracted, original system.
3. If the check returns "unknown", the abstraction needs to be refined. We either add a new predicate, or take one process out of the shade into the spotlight. Then we proceed with step 2.

Note that $P_\perp$ only modifies predicates about *global* variables. Thus, any set of shade processes which share the same operations on global variables V_g will give us the same approximative process $P_\perp^{V_g}$. For our semaphore example and the predicate $(y = 1)$ an arbitrary number of processes in the shade would be automatically abstracted into

$$P_\perp^{\{y\}} :: \begin{bmatrix} \texttt{loop forever do} & \\ \quad (y=1) := \begin{cases} \mathit{false} & \texttt{if } (y=1) = \mathit{false} \\ \mathit{unknown} & \texttt{else} \end{cases} \end{bmatrix}$$

approximating the possible operations `request` and `release` on the global semaphore variable y.

For employing 3Spot in our verification procedure for parametric systems, we do not only have to instantiate process variables in the formula, but we also have to fix the number n of processes in $||_{i=1}^{n} P_i$. This number is set to $d+1$, i.e. one more than the process diameter of the formula, in the case of formula F_1 thus to 3. Now we check the following property with 3Spot

$$P_1 \mid\mid P_2 \mid\mid P_3 \models AG\neg(1@2 \wedge 2@2) \ ?$$

Since the property is only referring to processes 1 and 2, the abstraction starts with 1 and 2 in the spotlight. It turns out that throughout the whole model checking procedure process 3 is kept in the shade and the verification succeeds with a definite answer (yes) without ever considering process 3. More specifically, 3Spot shows the following to be true:

$$P_1 \mid\mid P_2 \mid\mid P_\perp^{\{y\}} \models AG\neg(1@2 \wedge 2@2)$$

Here, $P_\perp^{\{y\}}$ is the abstraction of process 3. As explained above this is also the abstraction of *any* number of parallel processes performing request and release operations on the global variable y. The correctness result for spotlight abstraction now lets us transfer this result to the original parallel system.

Theorem 2. *Let* $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ *be a spotlight abstraction of a symmetric system for which checking property* $\psi(1,\ldots,d)$ *yields true. Then for any* $N > d$ *we get*

$$||_{i=1}^{N}\ P_i \models \psi(1,\ldots,d)\ .$$

Proof: In [17] we have shown that if $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ is a spotlight abstraction of a parallel system with a *fixed* number of processes and $\psi(1,\ldots,d)$ yields true for $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ then we can transfer this result to the unabstracted system. Here, we consider *symmetric* systems of an *arbitrary* size. Due to symmetry, the approximative process $P_\perp^{V_g}$ and therefore the entire spotlight abstraction is the same for *any* number $N > d$ of processes. □

In the next step, we need to transfer the result for particular process identities to arbitrary process variables. Theorem 1 exactly allows us to do this: verification results for $(1,\ldots,d)$ also hold for all permutations of $(1,\ldots,d)$. Since all possible instantiations of process variables i_1 to i_d can be obtained this way (note that we required all variable values to be pairwise different), the verification result also holds in general.

Corollary 1. *Let* $(||_{i=1}^{d}\ P_i) \mid\mid P_\perp^{V_g}$ *be a spotlight abstraction of a symmetric system for which checking property* $\psi(1,,\ldots,d)$ *yields true. Then the following holds:*

$$\forall\, N > d : ||_{i=1}^{N}\ P_i \models \forall\, i_1,\ldots,i_d, (i_l \neq i_j)_{1\leq l,j\leq d, l\neq j} : \psi(i_1,\ldots,i_d)$$

An analogue result is achieved for negative outcomes of the model checking runs: outcome "false" can also be transferred to the full system. In summary, we have the following verification steps: For a symmetric parameterised system $||_{i=1}^{n}\ P_i$ and a formula F execute the following steps for checking $\forall\, N : ||_{i=1}^{N}\ P_i \models F$:

1. Determine the process diameter d of F.
2. Instantiate F with process identities 1 to d for i_1 to i_d: $\psi(1,\ldots,d)$.
3. Check $P_1 \mid\mid \ldots \mid\mid P_d \mid\mid P_{d+1} \models \psi(1,\ldots,d)$.
4. If the result is "yes"/"no" and P_{d+1} is *not* in the spotlight in the final abstraction, return *true*/*false*. Else return *unknown*.

The obtained result is by symmetry then a valid result for the parameterised system. In this manner we can also disprove the liveness property F_2 by having only two processes in the spotlight.

3Spot might take the process P_{d+1} into the spotlight. In this case a definite outcome cannot be transferred to the full system because then there is no approximative process representing an arbitrary number of additional process instances. Hence, it is *unknown* whether ψ holds for *all* instantiations of the parameterised system. So this approach gives us a fully automatic, sound but not complete procedure for reasoning about parameterised, symmetric systems.

5 Generalisation

Our approach can be generalised to *classwise symmetric systems*. Such systems are not fully symmetric, but they consist of classes of symmetric processes. An example for a system consisting of two classes is the following solution to the readers/writers problem:

$$\texttt{global}\ y:\ \texttt{sem where}\ y = n_{Rd}$$

$$\Big\|_{i \in PID^{Rd}} Reader_i :: \begin{bmatrix} \texttt{loop forever do} \\ \begin{bmatrix} 0: & \texttt{Non-Critical} \\ 1: & \texttt{request}\ (y,1); \\ 2: & \texttt{Critical-Write} \\ 3: & \texttt{release}\ (y,1); \end{bmatrix} \end{bmatrix}$$

$$\Big\|$$

$$\Big\|_{j \in PID^{Wrt}} Writer_j :: \begin{bmatrix} \texttt{loop forever do} \\ \begin{bmatrix} 0: & \texttt{Non-Critical} \\ 1: & \texttt{request}\ (y,n_{Rd}); \\ 2: & \texttt{Critical-Write} \\ 3: & \texttt{release}\ (y,n_{Rd}); \end{bmatrix} \end{bmatrix}$$

Here, we have two classes of processes, readers and writers. Furthermore, n_{Rd} is a parameter associated with the number of reader processes. The semantics of the semaphore statements `request` (y, c) and `release` (y, c) are given by $\langle \texttt{await}\ y \geq c;\ y := y - c \rangle$ and $\langle y := y + c \rangle$, respectively. Thus, the generalised semaphore y ensures that multiple (up to n_{Rd}) readers can enter the critical section at the same time, whereas if one writer is modifying data, no other process has access to the critical section. All readers and all writers execute the same program code. More generally, a classwise symmetric system consisting of k classes is defined as $P = ||_{m=1}^{k} P^m$ where $P^m = ||_{i \in PID^m} P_i$ is a parallel composition of fully symmetric processes of a class m with process identifiers from a set PID^m. All sets PID^m, where $m \in [1..k]$, are pairwise disjoint. Thus,

every process in a classwise symmetric system has a unique id. All classes share a set of global variables V_g whereas each class has a distinct set of local variables V_l^m with $\forall\, m_1, m_2 \in [1..k] : V_l^{m_1}, V_l^{m_2}$ pairwise disjoint. Furthermore, each V_l^m contains a dedicated program counter pc^m. The overall set of variables of P is $V = V_g \cup \bigcup_{m=1}^{k}(V_l^m \times PID^m)$.

Given this setting, we can define the semantics in a way similar to fully symmetric systems. States of a classwise symmetric system are defined as mappings $s : V \rightarrow D$. The local view of a process P_i in a state s is again denoted by $s[i]$, referring to its unique identifier i. Transitions in such a system caused by the execution of a statement in some process P_i are described by a predicate R_i on the primed and unprimed local views of P_i. Due to the symmetry of processes in a class m, predicates R_i are the same for all $i \in PID^m$.

Classwise symmetric systems can be represented as Kripke structures as well. As atomic propositions we use the same as before: $(v = d)$, $(i@l)$ and $(v@i = d)$. For a system $P = ||_{m=1}^{k} P^m$ with $P^m = ||_{i \in PID^m} P_i$ the corresponding Kripke Structure $K = (S, s_0, R, L)$ is defined as follows:

- States: $S := V \rightarrow D$,
- Initial state: $s_0 := s \in S$ with $s \restriction V_g \models \varphi_{ginit} \wedge \forall\, m \in [1..k], \forall\, i \in PID^m$: $s[i] \models \varphi_{linit^m}$,
- Transition function:

$$\begin{aligned} R(s, s') := \exists\, m \in [1..k], \exists\, i \in PID^m &: R_i(s[i], s'[i]) \\ \wedge\ \forall\, i_2 \in PID^m, i_2 \neq i, \forall\, v \in V_l^m &: s[i_2](v) = s'[i_2](v) \\ \wedge\ \forall\, m_2 \neq m, \forall\, j \in PID^{m_2}, \forall\, v \in V_l^{m_2} &: s[j](v) = s'[j](v), \end{aligned}$$

- Labelling function:

$$L(s, p) := \begin{cases} s(v) = d & \text{for } \ p \mathrel{\hat{=}} (v = d) \\ s(pc^m, i) = l & \text{for } \ p \mathrel{\hat{=}} (i@l), i \in PID^m \\ s(v, i) = d & \text{for } \ p \mathrel{\hat{=}} (v@i = d), i \in PID^m, v \in V^m \end{cases}$$

For classwise symmetric systems $P = ||_{m=1}^{k} P^m$ we like to show properties that refer to distinct classes but arbitrary processes in each class. More precisely, a property formula takes the following form:

$$F = \forall\, i_1^1, \ldots, i_{d_1}^1, \ldots, \forall\, i_1^k, \ldots, i_{d_k}^k : \psi(i_1^1, \ldots, i_{d_1}^1, \ldots, i_1^k, \ldots, i_{d_k}^k)$$

where $\psi \in$ CTL and i_1^m to $i_{d_m}^m$ with $m \in [1..k]$ are pairwise different variables for process identifiers in PID^m. Thus, for every class m referred in F there is a distinct process diameter d_m. For our readers/writers example we have two classes of symmetric processes: *readers* with the corresponding set of process identifiers PID^{Rd} and *writers* with PID^{Wrt}. A safety property about mutual exclusion (no reading and writing at the same time, and no writing concurrently) can be formalised as: $F_3 : \forall\, i \in PID^{Rd}, \forall\, j_1, j_2 \in PID^{Wrt} : AG\neg(i@2 \wedge j_2@2) \wedge AG\neg(j_1@2 \wedge j_2@2)$ where the process diameter is 1 for readers and 2 for writers.

Since we like to show such properties for an arbitrary number of processes from each class running in parallel, we want exploit the symmetry again. As the considered systems are not fully but classwise symmetric, we need a new notion of process permutations.

Definition 6. *A* class-sensitive process permutation *is a bijective function* $\pi : \bigcup_{m=1}^{k} PID^m \rightarrow \bigcup_{m=1}^{k} PID^m$ *with* $i \in PID^m \Leftrightarrow \pi(i) \in PID^m$ *for all* $m \in [1..k]$ *and* $i \in PID^m$*.*

Class-sensitive permutations preserve the class affiliation of process identifiers. Hence, on classwise symmetric systems all class-sensitive process permutations are symmetries. Alike the fully symmetric case, we get the following result:

Theorem 3. *Let* $||_{m=1}^{k} P^m$ *be a classwise symmetric system,* $K = (S, s_0, R, L)$ *the corresponding Kripke structure over* AP *and* π *a class-sensitive process permutation which is a symmetry for* K*. Moreover, let* $s \in S$ *and* ψ *be a CTL formula referring to concrete process identifiers* $p_j^m \in PID^m$*,* $j \in [1..d_m]$*,* $m \in [1..k]$*. Then*

$$\begin{aligned} & K, s \models \psi(p_1^1, \ldots, p_{d_1}^1, \ldots, p_1^k, \ldots, p_{d_k}^k) \\ \Leftrightarrow\ & K, \pi(s) \models \psi(\pi(p_1^1), \ldots, \pi(p_{d_1}^1), \ldots, \pi(p_1^k), \ldots, \pi(p_{d_k}^k)) \ . \end{aligned}$$

Thus, applying class-sensitive permutations preserves the validity of CTL properties referring to *particular* processes of a classwise symmetric system. Via the spotlight technique we can extend this result to formulas referring to *arbitrary* processes. The process $P_\perp^{V_g}$ now approximates all operations on global variables occurring in *any* class of the considered system. Here, $PID_{d_m}^m$ denotes an arbitrary subset of PID^m with d_m elements.

Theorem 4. *Let* $(||_{m=1}^{k}||_{i \in PID_{d_m}^m} P_i) \,||\, P_\perp^{V_g}$ *be a spotlight abstraction of a classwise symmetric system for which checking property* $\psi(p_1^1, ..., p_{d_1}^1, ..., p_1^k, ..., p_{d_k}^k)$ *yields true. Then the following holds:*

$$\begin{aligned} & \forall\, N_1 > d_1, ..., N_k > d_k : \\ & ||_{m=1}^{k}||_{i \in PID_{N_m}^m} P_i \models \forall\, i_1^1, ..., i_{d_1}^1, ..., \forall\, i_1^k, ..., i_{d_k}^k : \psi(i_1^1, ..., i_{d_1}^1, ..., i_1^k, ..., i_{d_k}^k) \end{aligned}$$

As before, negative outcomes can be transferred to the original system as well. This result lets us exploit once again symmetries to verify parameterised systems by using spotlight abstractions; now for a generalised notion of symmetric systems, called classwise symmetric systems. For our readers/writers example we can prove the safety property F_3 via 3Spot by taking just one reader process and two writer processes into the spotlight. The verification was performed in $0.83s$ (on a 2.40GHz Core 2 Duo Windows system with 3GB memory).

6 Conclusion

In this paper we have proposed a verification technique for parameterised systems. It provides for a sound proof technique for full CTL properties by a combination

of symmetry reduction and spotlight abstraction. In case that the spotlight abstraction draws too many processes into the spotlight, the validity of properties for the parameterised system is however left open. In the future we therefore intend to investigate whether this technique can give us more results when we use larger instantiations, i.e. larger than "1 plus process diameter". This might be essential for reasoning about several liveness properties: the formula $F_4 : \forall\, i : AG(i@1 \Rightarrow AF(i@2))$ refers to a single process only. Thus, the process diameter is one. However, for proving or disproving starvation freedom it is obviously necessary to have at least one "adversary" process in the spotlight and therefore an instantiation larger than $d+1$. Further investigation might allow us to find appropriate instantiation sizes for distinct types of temporal logic formulae. Moreover, our approach could be easily modified to an iterative but possibly infinitely running procedure: We gradually add processes $P_{d+2}, P_{d+3}, \ldots$ to the instantiation until we get a definite answer without having all process instances in the spotlight. This might not terminate in all cases (e.g. when the property cannot be proven on a finite-state abstraction). Anyway, the formula F_4 can be disproved for the mutual exclusion example in this way within two iterations and $0.25s$.

Related work. The generally undecidable problem of parameterised verification has received a lot of attention in research. Several approaches (e.g. [8,5,7,14]) try to bypass this problem by summarising the considered system by a finite instantiation, based on symmetry arguments. From Clarke et al. [5] we have taken the idea of exploiting symmetry under permutations. In our work permutations are furthermore applied to atomic propositions which is also done in [11]. The work most closely related to ours is that of Emerson and Kahlon [7] who reduce reasoning for parameterised systems of an arbitrary size to systems of a small cutoff size. Contrary to the process diameter in our technique the cutoff size does not depend on the property but on the description of the system. Moreover, their method is complete but restricted to processes of a certain structure and properties in a fragment of CTL$^*\backslash$X. Similar to [7], Namjoshi [14] proposes a cutoff-based verification technique for parameterised systems with less restrictions to the structure of the processes but limited to safety properties.

The principle of spotlight abstraction was first introduced by Wachter and Westphal [18] and later enriched with three-valued semantics in our previous work [17]. To the best of our knowledge, we are the first to combine symmetry reduction and spotlight abstraction in parameterised model checking. In former approaches symmetry reduction has been used in combination with partial order reduction [6] and heuristic search [12].

Acknowledgement. We thank Daniel Wonisch for help with extending 3Spot.

References

1. Abdulla, P.A., Delzanno, G., Henda, N.B., Rezine, A.: Monotonic abstraction: on efficient verification of parameterized systems. Int. J. Found. Comput. Sci. 20(5), 779–801 (2009)

2. Abdulla, P.A., Jonsson, B., Nilsson, M., Saksena, M.: A survey of regular model checking. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 35–48. Springer, Heidelberg (2004)
3. Apt, K.R., Kozen, D.: Limits for automatic verification of finite-state concurrent systems. Inf. Process. Lett. 22(6), 307–309 (1986)
4. Baukus, K., Lakhnech, Y., Stahl, K.: Parameterized verification of a cache coherence protocol: Safety and liveness. In: Cortesi, A. (ed.) VMCAI 2002. LNCS, vol. 2294, pp. 317–330. Springer, Heidelberg (2002)
5. Clarke, E.M., Jha, S., Enders, R., Filkorn, T.: Exploiting symmetry in temporal logic model checking. Formal Methods in System Design 9(1/2), 77–104 (1996)
6. Emerson, E., Jha, S., Peled, D.: Combining partial order and symmetry reductions. In: Brinksma, E. (ed.) TACAS 1997. LNCS, vol. 1217, pp. 19–34. Springer, Heidelberg (1997)
7. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
8. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
9. Fitting, M.: Kleene's three valued logics and their children. Fundamenta Informaticae 20(1-3), 113–131 (1994)
10. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design 9(1/2), 41–75 (1996)
11. Leuschel, M., Butler, M.J., Spermann, C., Turner, E.: Symmetry reduction for b by permutation flooding. In: B, pp. 79–93 (2007)
12. Lluch-Lafuente, A.: Symmetry reduction and heuristic search for error detection in model checking. In: 2nd Workshop on Model Checking and Artificial Intelligence (MoChArt), pp. 77–86 (2003)
13. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
14. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
15. Pnueli, A., Ruah, S., Zuck, L.D.: Automatic deductive verification with invisible invariants. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 82–97. Springer, Heidelberg (2001)
16. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, infinity)-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
17. Schrieb, J., Wehrheim, H., Wonisch, D.: Three-valued spotlight abstractions. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 106–122. Springer, Heidelberg (2009)
18. Wachter, B., Westphal, B.: The spotlight principle. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 182–198. Springer, Heidelberg (2007)

A Methodology for Automatic Diagnosability Analysis

Jonathan Ezekiel and Alessio Lomuscio

Department of Computing, Imperial College London, UK
{jezekiel,alessio}@doc.ic.ac.uk

Abstract. We present an algorithm based on temporal-epistemic model checking combined with fault injection to analyse automatically the diagnosability of faults by agents in the system. We describe an implementation built on the multi-agent systems model checker MCMAS and a dedicated compiler for injecting faults into an MCMAS program. A diagnosability report is generated by the implementation which can be utilised at an early stage of fault tolerant multi-agent system design to ensure accurate fault diagnosis. We demonstrate the practical usefulness of the algorithm by performing automatic diagnosability analysis on a model of the IEEE 802.5 token ring LAN protocol which employs fault diagnosis mechanisms to achieve fault tolerance.

1 Introduction

Distributed fault tolerant systems are notoriously difficult to understand and design due to their high level of complexity [10]. A potential way to manage this complexity is offered by the multi-agent systems (MAS) paradigm [25] in which agents, representing processes of a distributed system, autonomously interact with one another, engaging in communication, co-ordination, negotiation, etc. Moreover, a number of fault tolerant MAS architectures have been designed which utilise strategies such as agents *diagnosing* faults so that they can communicate and co-ordinate to recover from them (see e.g., [18]).

As a design paradigm, MAS has many applications including, but not limited to distributed control systems (DCS) (see e.g., [20]). Within the general area of DCS it is known that safe design is a major industrial concern since DCS are becoming increasingly complex and involved in many safety-critical applications [6]. However, practical approaches towards *verifying* fault tolerant MAS are required to certify that they conform to the stringent requirement of operating correctly under degraded conditions [1].

Recently, an approach combining fault injection [16] with model checking [8] has been used to verify the correctness of fault tolerance mechanisms in reactive systems [2, 4, 5, 17] and MAS [12, 13]. In contrast to ad-hoc modelling of faulty behaviour, in this approach faults can be automatically injected into a model of a correctly behaving system to create a mutated model which exhibits both correct and faulty behaviour. Temporal-epistemic specifications [14] can then be verified to analyse the correct and faulty behaviour of agents in the mutated model, as

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 549–564, 2010.
© Springer-Verlag Berlin Heidelberg 2010

well as the knowledge that agents have about the behaviour. This allows for the verification of fault tolerance, recovery from faults, and *diagnosability*, i.e., whether an unobservable fault can be accurately diagnosed from the observable events of the system [22].

The high level of usability offered by the automatic nature of both the fault injection and the model checking process makes the approach particularly attractive to non-experts in verification [4]. Another advantage is the ability to use the model checker to generate automatically artifacts that analyse the impact of the injected faults such as fault trees [23]. To date, these artifacts have been generated by using temporal formulas to describe which component failures occur as a result of the injected faults [3, 4]. However, artifacts relating to diagnosability have yet to be suggested in the literature.

In this paper we show how a diagnosability artifact can be generated from a combined fault injection and temporal-epistemic model checking [15, 19, 21] approach by presenting a methodology for automatic *diagnosability analysis*. The analysis is used to provide the user with a report on the diagnosis of each injected fault by every agent in the system. We consider this to be part of a practical approach towards verifying fault tolerant MAS that can be used at an early stage of system design to ensure that agents in the system accurately diagnose faults.

We implement these ideas by integrating the algorithm we propose with the model checker MCMAS [19] and an existing fault injection compiler that injects automatically faults into a model for input into MCMAS [12, 13]. We describe a framework in which powerful modules which generate fault analysis artifacts for fault tolerant MAS can be integrated with MCMAS and the fault injection compiler. To highlight the practical usefulness of the algorithm from a user perspective we use the implementation to perform automatic diagnosability analysis on a model of the IEEE 802.5 token ring LAN protocol from [13] which utilises distributed diagnosis mechanisms to achieve fault tolerance.

The rest of the paper is structured as follows. In Section 2 we provide the background on model checking, interpreted systems, MCMAS, fault injection, and artifact generation. In Section 3 we present the algorithm for diagnosability analysis. In Section 4 we describe a framework for integrating fault analysis modules with MCMAS and the fault injection compiler, which we use to implement the algorithm for diagnosability analysis. In Section 5 we show how the diagnosability analysis module is applied to the token ring protocol. In Section 6 we discuss the related work and in Section 7 we conclude and put forward future work.

2 Background

Model checking [8] is a widely adopted technique for systems verification. The system considered for verification S is represented by a logical model M_S which encodes the behaviour of the system as computational traces. A specification of a property P is expressed by means of a logical formula φ_P. The model checker establishes whether or not M_S satisfies φ_P (formally, $M \vDash \varphi_P$). The

satisfaction relation is implemented as an automatic decision procedure, making model checking attractive for the purpose of verification [8]. In the case of MAS φ_P is often expressed by using a number of rich modal logics including temporal, ATL, and epistemic logics [25]. Particularly relevant to diagnosability is temporal-epistemic logic, which can be used to reason about the *knowledge* of the agents over time.

2.1 Interpreted Systems and MCMAS

We summarise the key points of the formalism used by following the presentation given in [12]. Interpreted systems [14] are a popular semantics for temporal-epistemic logic. Each agent $i \in \{1, \ldots, n\}$ in the system is characterised by a finite set of local states L_i and by a finite set of actions Act_i. Actions are performed in compliance with a protocol $P_i : L_i \rightarrow 2^{Act_i}$, specifying which actions may be performed in a given state. In this formalism the environment in which agents live may be modelled by means of a special agent E. Associated with E are a set of local states L_E, a set of actions Act_E, and a protocol P_E. A tuple $g = (l_1, \ldots, l_n, l_E) \in L_1 \times \ldots \times L_n \times L_E$ where $l_i \in L_i$ for each i and $l_E \in L_E$, is a *global state* describing the system at a particular instant of time.

The evolution of the agents' local states is described by a function $t_i : L_i \times L_E \times Act_1 \times \cdots \times Act_n \times \cdots Act_E \rightarrow L_i$, which returns a local state (the next local state) for agent i given the current local state of the agent, the current local state of the environment and all the agents' actions. Similarly the evolution of the environment's local states is described by a function $t_E : L_E \times Act_1 \times \cdots \times Act_n \times \cdots Act_E \rightarrow L_E$. It is assumed that in every state agents evolve simultaneously. The evolution of the global states of the system is described by a function $t : S \times Act \rightarrow S$, where $S \subseteq L_1 \times \cdots \times L_n \times L_E$, and $Act \subseteq Act_1 \times \cdots \times Act_n \times Act_E$. The function t is defined by $t(g,a) = g'$ iff for all $i, t_i(l_i(g), a) = l_i(g')$ and $t_E(l_E(g), a) = l_E(g')$, where $l_i(g)$ denotes the i-th component of a global state g (corresponding to the local state of agent i). Given a set $I \subseteq S$ of possible initial global states, a set $G \subseteq S$ of reachable global states is generated by all possible runs of the system. Finally, the definition includes a set of atomic propositions AP together with a valuation function $V : S \rightarrow 2^{AP}$. We define an *interpreted system* as the tuple:

$$IS = \langle (L_i, Act_i, P_i, t_i)_{i \in \{1,\ldots,n\}}, (L_E, Act_E, P_E, t_E), I, V \rangle$$

The syntactical constructs and the semantic model that are presented in [19] are adopted for the interpretation of temporal-epistemic formulae in interpreted systems. Specifically, we consider the following syntax defining our specification language:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid EX\varphi \mid AG\varphi \mid E(\varphi U \varphi) \mid K_i\varphi$$

In the grammar above $p \in AP$ is an atomic proposition; EX is a temporal operator expressing that there exists a next state in which φ holds; AG is a temporal operator expressing that in all runs φ holds globally; $E(\varphi U \psi)$ is a temporal operator expressing that there exists a run in which φ holds until ψ holds; $K_i\varphi$ expresses that *agent i knows* φ [14].

IS is associated with a model $M_{IS} = (W, R_t, \sim_1, \ldots, \sim_n, L)$ that can be used to interpret any formula φ. The set of possible worlds W is the set G of reachable global states. The temporal relation $R_t \subseteq W \times W$ relating two worlds (i.e., two global states) is defined by considering the temporal transition t. Two worlds w and w' are such that $R_t(w, w')$ iff there exists a joint action $a \in Act$ such that $t(w, a) = w'$, where t is the transition relation of IS. The epistemic accessibility relations $\sim_i \subseteq W \times W$ are defined by considering the equality of the local components of the global states. Two worlds w,$w' \in W$ are such that $w \sim_i w'$ iff $l_i(w) = l_i(w')$ (i.e., two worlds w and w' are related via the epistemic relation $\sim_i$ when the local states of agent i in global states w and w' are the same [14]). The labelling relation $L \subseteq AP \times W$ can easily be defined in terms of the valuation relation V.

Formulae can be interpreted in M_{IS} in a standard way [14] as follows. Let $\pi = (w_0, w_1, \cdots)$ be an infinite sequence of global states such that for all i, $R_t(w_i, w_{i+1})$, and let $\pi(i)$ denote the i-th world of the sequence (notice that, following standard conventions we assume that the temporal relation is serial and thus all computation paths are infinite). We write $(M, w) \models \varphi$ to represent that a formula φ is true at a world w in a Kripke model M, associated with an interpreted system IS. Satisfaction is defined as follows.

$(M, w) \models p$ iff $(p, w) \in L$;
$(M, w) \models \neg\varphi$ iff it is not the case that $M \models \varphi$;
$(M, w) \models \varphi_1 \vee \varphi_2$ iff either $M \models \varphi_1$ or $M \models \varphi_2$;
$(M, w) \models EX\varphi$ iff there exists a path π such that $\pi(0) = w$, and $(M, \pi(1)) \models \varphi$;
$(M, w) \models AG\varphi$ iff for all paths π such that $\pi(0) = w$ we have that $(M, \pi(i)) \models \varphi$, for all $i \geq 0$;
$(M, w) \models E(\varphi U \psi)$ iff there exists a path π such that $\pi(0) = w$, and there exists $k \geq 0$ such that $(M, \pi(k)) \models \psi$ and $(M, \pi(j)) \models \varphi$ for all $0 \leq j < k$;
$(M, w) \models K_i\varphi$ iff for all $w' \in W$, $w \sim_i w'$ implies $(M, w') \models \varphi$.

We say that a formula φ is true in the model, and we write $M \models \varphi$, if $(M, w) \models \varphi$ for all $w \in W$. Similarly to [14], we say that a formula φ is true in an interpreted system IS, and we write $IS \models \varphi$, if $M_{IS} \models \varphi$. *A formula is true in an interpreted system if it is true in the associated Kripke model.*

MCMAS [19] provides ISPL as an input language for modelling a MAS and expressing (amongst others) temporal and epistemic formulas as specifications of the system. ISPL programs are closely related to interpreted systems; specifically each ISPL program describes an interpreted system. MCMAS supports the verification for all formulas in the language above. The structure of an ISPL program allows the local states to be defined using *boolean*, *bounded integer*, and *enumeration* variables.

2.2 Fault Injection into MAS Programs

The first step of a combined fault injection and model checking approach [2, 4, 5, 13, 17] involves *mutating* a model of correct system behaviour by injecting

faulty behaviour into it. The output of the mutation step is a model containing correct and faulty behaviour.

We summarise the mutation technique for interpreted systems defined in [12, 13]. Any agent A of the system can be mutated into a faulty agent A^{F*} which includes the faulty behaviour that results from a fault occurring. For each fault an additional fault injection agent FI implements the timing characteristics of the fault. The faulty behaviour is triggered in the faulty agent whenever an $inject$ action is performed by FI. Conversely, the correct behaviour is preserved in the faulty agent whenever the $inject$ action is not performed by FI.

The faulty behaviour in the faulty agent A^{F*} is introduced using a number of *mutation rules* which determine how the evolution function t_A is mutated to $t_{A^{F*}}$. A *variable value replace* fault defines that the value of a variable var is updated with a value v_2 in $t_{A^{F*}}$ whenever the value of var is updated to a value v_1 in t_A. This fault is useful for defining faulty conditions where some of the correct agent behaviour is skipped. A *stuck at select* fault defines that the value v_1 of a variable var persists if the current value of var is v_1. If in t_A the variable var is updated to a value $v_x \neq v_1$ when $var = v_1$, the faulty behaviour in $t_{A^{F*}}$ preserves $var = v_1$. This fault can be used to define behaviour in which a component becomes stuck in particular state. Further rules are defined in [12].

The occurrence of the $inject$ action is determined by the behaviour of the fault injection agent FI. A number of timing options can be selected for FI: injecting *constantly*, *randomly*, *after a random start*, *until a random stop*, and *after and until an action has occurred* [12]. The local states, actions, protocol, and evolution function of the fault injection agent are defined according to these options, which can be combined to create complex timing characteristics of the fault. A mutated set of initial states I^{F*} stipulates that the local state of FI is set to either $notfaulty$ which persists throughout the system run, or to a state in which faults may be injected into the system by FI in the future according to the timing options.

A mutated valuation function V^{F*} relates atomic propositions to the local states of each fault injection agent. This can be used to reason about the correct and faulty behaviours of the mutated interpreted system IS^{F*}. For each fault $j \in \{1, \ldots, m\}$ the mutated set of atomic propositions AP^{F*} extends with the propositions $faulty_j$, $injected_j$, $injecting_j$, $stopped_j$. $faulty_j$ represents that a fault can be injected during the system run; $injected_j$ expresses that a fault is injected at the current tick of the clock (i.e., a global state describing the system at a particular instant of time); $injecting_j$ denotes that a fault can be injected at the current tick of the clock; $stopped_j$ describes that a fault has been injected but can no longer be injected at the current tick of the clock. The extended faulty system is defined as:

$$IS^{F*} = \langle (L^{F*}{}_i, Act^{F*}{}_i, P^{F*}{}_i, t^{F*}{}_i)_{i \in \{1,\ldots,n+m\}}, (L_E, Act_E, P_E, t_E), I^{F*}, V^{F*} \rangle$$

Once a mutated model IS^{F*} has been obtained, both the correct and faulty behaviours of the system can be analysed. A library of *specification patterns* pertaining to fault tolerance, recoverability, and diagnosability defined in [12, 13] can be used to reason about properties of the system in relation to the taxonomy

of dependable computing given in in [1]. Such properties include whether the fault becomes an error that is propagated internally amongst the agents of the system, whether it can be diagnosed, recovered from, or further propagated to the service interface to cause a failure. In this case, the *system boundary* is the shared actions between the agents of the system and the environment. Thus, we can verify properties of the system that affect the system boundary under faulty behaviour to determine whether the fault becomes a failure.

We highlight two specification patterns that are relevant to this paper. To reason about fault tolerance for a property ϕ we can analyse [13]:

$$AG((\neg faulty_j \wedge faulty_k) \rightarrow \phi) \quad (1)$$

This formula states that ϕ always holds whenever fault j is never injected into the model and fault k may be injected into the model. The formula specifies the ability of the system to tolerate faults, in this instance, fault k.

Usually diagnosability is informally defined by saying that *a fault is diagnosable if some observations after the occurrence of the fault can correctly identify it* [22]. A fault is diagnosable if any agent of the system *knows* about it at some point after its occurrence. We can express this as [12]:

$$\neg E(\neg injected_j\ U\ (injected_j \wedge \neg AF(K_i(injecting_j \vee stopped_j)))) \quad (2)$$

This formula states that there is no path in which at some point fault j is injected and from that time it is not true that at some point in the future agent i knows fault j can be or has been injected. In other words the formula specifies the ability of agent i to diagnose fault j correctly after j has first been injected.

2.3 Generating Fault Analysis Artifacts

A particular advantage of using an approach based on model checking and fault injection is the ability to generate automatically artifacts such as fault trees [23]. A fault tree is a graphical representation which is constructed by identifying a *minimal cut set* of events that can cause a system malfunction to occur. This malfunction is also known as a top level event (TLE). The fault tree displays these events connected by logic gates to highlight their relation to each other.

The process of identifying this minimal cut set can be automated by combining fault injection and model checking [3]. The function $h : \{f, \neg f, \epsilon\}^m \rightarrow \{T, F\}$ represents the *cut set* for m injected faults where f corresponds to the fault being injected, $\neg f$ corresponds to the fault not being injected, and ϵ corresponds to the fault being either injected or not injected. For example, $h(\{f_1, \epsilon, \neg f_3\}) = T$ represents that the TLE occurs when: fault 1 is injected, fault 2 may or may not be injected, and fault 3 is not injected. In other words a TLE occurs when fault 1 is injected and fault 3 is not irrespective of fault 2. The cut set can be created automatically by using a model checker to verify a number of specifications against a mutated model in order to determine whether a TLE occurs for each fault injection combination of the cut set. Formula 1 is an example of a suitable formula that can be used to define specifications that create the cut set.

Once the cut set has been created the *prime implicants* of the cut set, i.e., the minimal set of events relevant to the occurrence of the TLE can be determined. For example, a cut set $f_1 \vee (f_1 \wedge f_3)$ describes that when fault 1 is injected, or when fault 1 is injected and fault 3 is injected, the TLE occurs. In this cut set f_1 is a prime implicant representing the minimal cut set, since f_1 is sufficient for all occurrences of the TLE. To identify a minimal cut set, many prime implicant algorithms exist, for further discussion see [3].

Fault trees are useful for studying the impact of faults, however, they provide limited insight into the cause of system malfunctions. At an early stage of design it is desirable to analyse diagnosability so that system malfunctions do not occur as a result of fault tolerance mechanisms diagnosing faults inaccurately.

3 Diagnosability Analysis

In this section we apply the general ideas for artifact generation presented in the previous section in order to analyse diagnosability. We achieve this by employing a diagnosability formula to generate a diagnosability cut set and by defining an algorithm to identity a minimal cut set. Unlike fault trees which communicate the impact of faults, there are no pre-defined graphical representations of diagnosability. Thus, any diagnosability specification pattern used for analysis must be carefully selected in order to generate a cut set of meaningful events. As a starting point for producing diagnosability artifacts we choose Formula 2. The formula is used to determine the diagnosis of faults by each of the non fault-injection agents in the system.

In the following we define the diagnosability cut set. For each non fault-injection agent $i \in \{1, \ldots, n\}$ and each fault $j \in \{1, \ldots, m\}$, given a *diagnosis group*, i.e., a group of faults for diagnosis $DG \in 2^{\{1,\ldots,m\}}$, the cut set which determines that a diagnosis group can be diagnosed by an agent after a fault has first been injected into the system is described by the function $D : \{1, \ldots, n\} \times \{1, \ldots, m\} \times 2^{\{1,\ldots,m\}} \rightarrow \{T, F\}$, where T indicates that the diagnosis can be made. For example $D(i, j, \{j, k\})$ represents whether agent i diagnoses that either fault j or fault k has occurred after fault j is first injected into the system. We can define this function where $ijst_k = (injecting_k \vee stopped_k)$ using Formula 2 as follows:

$$\begin{aligned} &D(i, j, DG) = T \text{ iff } j \in DG \text{ and} \\ &\qquad IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \;\wedge\; \neg AF(K_i(\textstyle\bigvee_{k \in DG} ijst_k)))) \end{aligned}$$

In other words the diagnosability cut set function evaluates to true if fault j is in the diagnosis group DG and the mutated interpreted system IS^{F*} is satisfied by a formula describing that agent i diagnoses the faults in DG after j has first been injected.

Due to the way that the mutated initial states are defined in IS^{F*} there is at least one path in IS^{F*} in which at some point fault j is injected and along that path any fault $k \in \{1, \ldots, m\} \setminus \{j\}$ is never injected. Agent i cannot know about fault k along a path in which k is never injected. Thus if:

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\bigvee_{k \in DG} ijst_k))))$$

it follows that:

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\bigvee_{k \in DG} ijst_k) \wedge \neg K_i(\bigvee_{k \in DG \setminus \{j\}} ijst_k))))$$

The formula above specifies in addition to Formula 2 that agent i does not know that any of the faults in DG other than j can be or has been injected into the system. Given that this specification is implied by Formula 2 it is implicitly included in the definition of the diagnosability cut set function.

For each fault injected into the system the minimal cut set must contain the *most specific diagnosis* of that fault by each of the agents, i.e., if an agent can diagnose a fault that has been injected into the system, then it is meaningful to identify the smallest set of faults that the injected fault is part of, that can be diagnosed by the agent. For example, in a network protocol it may be necessary to diagnose a specific severe fault in order to reconfigure a router to bypass a workstation. Conversely, it may suffice to diagnose that any of a number of possible intermittent faults have occurred so that a message can be resent.

The diagnosability cut set function D does not determine the most specific diagnosis. This is because if DG can be diagnosed by an agent, then according to the conjunction of faults in the definition of Formula 2, $DG \cup DH \in 2^{\{1,\dots,m\}}$ can be diagnosed by the agent. For example, given three faults j, k, and l, if $D(i,j,\{j,k\}) = T$, then $D(i,j,\{j,k,l\}) = T$, but only $\{j,k\}$ should be included in the minimal cut set. Thus, we wish to identify a minimal cut set $MCS_{ij} \subset 2^{\{1,\dots,m\}}$ containing the most specific diagnosis of fault j by agent i. To identify the minimal cut set we only need to consider conjunctions. Instead of employing a prime implicant algorithm that considers disjunctions we define a restriction to the minimal cut set as follows:

$$DH \in 2^{\{1,\dots,m\}} \notin MCS_{ij} \text{ iff } D(i,j,DG \in 2^{\{1,\dots,m\}}) = T \text{ and } DG \subset DH$$

In other words for every diagnosis group in the minimal cut set there can not be any subsets of that diagnosis group in the minimal cut set.

If we apply this restriction to the minimal cut set then for every diagnosis group $DY \in MCS_{ij}$ we have that for every diagnosis group $DZ \subset DY$, $D(i,j,DZ) = F$. It follows that:

$$IS^{F*} \not\models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(\bigvee_{DZ}^{DY} K_i(\bigvee_{k \in DZ} ijst_k))))$$

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(\bigwedge_{DZ}^{DY} \neg K_i(\bigvee_{k \in DZ} ijst_k))))$$

hence:

$$IS^{F*} \models \neg E(\neg injected_j \; U \; (injected_j \; \wedge \; \neg AF(K_i(\bigvee_{k \in DY} ijst_k) \bigwedge_{DZ}^{DY} \neg K_i(\bigvee_{k \in DZ} ijst_k))))$$

```
IdentifyMCS(in i:agentid, j:faultid, m:int) out MCS ⊂ 2^{1,...,m}
  [Identify and return a minimal cut set MCS for agent i and fault j given m faults.]

  X = 2^{1,...,m}
  DG ⊂ 2^{1,...,m}
  l: int
  1.  for l = 1 to m do
  2.    for each DG ∈ X where (j ∈ DG and |DG| = l) do
  3.      if (NotSubsetInMCS(DG, MCS) = true and D(i, j, DG) = T)
  4.          MCS = MCS ∪ DG
  5.  return MCS

NotSubsetInMCS(in DG ⊂ 2^{1,...,m}, MCS ⊂ 2^{1,...,m}) out bool
  [Return true if any element of MCS is a subset of DG otherwise return false.]

  DY ⊂ 2^{1,...,m}
  1.  for each DY ∈ MCS do
  2.    if DY ⊂ DG
  3.      return false
  4.  return true
```

Fig. 1. Psuedo-code for the diagnosability analysis algorithm

where $\bigwedge_{DZ}^{DY}$ and $\bigvee_{DZ}^{DY}$ are the conjunctions and disjunctions of all subsets of indices $DZ \subset DY$ respectively. The restriction defined for the minimal cut set determines that every diagnosis group in the minimal cut set is verified by the strong specification above. This specifies, in addition to Formula 2, that agent i does not know that any combination of the faults in every diagnosis group $DZ \subset DY$ other than j can be or has been injected into the system.

Based on the cut set function D and the minimal cut set restriction, we defined a sequential algorithm that performs diagnosability analysis to identify a minimal cut set for each agent as shown in Figure 1. A description for both the functions in the algorithm is as follows:

IdentifyMCS: identifies a minimal cut set for agent i and fault j which are passed as arguments to $IdentifyMCS$. Lines 1-2 define the iteration through all the possible diagnosis groups that include fault j, in ascending order of size. The iteration order ensures that the restriction function only needs to be applied once to each diagnosis group. Line 3 checks that a diagnosis group DG does not have any subsets in the minimal cut set and whether the faults in DG can be diagnosed for agent i and fault j. If both these conditions are met then the diagnosis group is added to the minimal cut set. Line 6 returns the minimal cut set for agent i and fault j.

NotSubsetInMCS: applies the restriction to the minimal cut set by checking to see whether any subsets of a diagnosis group DG passed as an argument to $NotSubsetInMCS$ are subsets of any diagnosis groups in the minimal cut set MCS passed as an argument to $NotSubsetInMCS$. Lines 1-3 iterate through all the diagnosis groups DY in MCS and $NotSubsetInMCS$ returns false if any

diagnosis group DY is a subset of DG. Line 4 is reached if no diagnosis group DY in MCS is a subset of DG, at which point $NotSubsetInMCS$ returns true.

Example: Given a system with four faults j, k, l, and m, in which agent i diagnoses that either faults j or k have occurred, or faults j or l or m have occurred, after fault j has been injected, a call to $IdentifyMCS(i, j, 4)$ performs four iterations of the for loop in line 2. *Iteration 1*: Diagnosis groups containing one element that include fault j: $\{j\}$. Since MCS is empty $NotSubsetInMCS$ always returns true. The function $D(i, j, \{j\})$ returns false and $\{j\}$ is not added to MCS. $MCS = \emptyset$. *Iteration 2*: Diagnosis groups which contain two elements that include fault j: $\{j, k\}, \{j, l\}, \{j, m\}$. Since MCS is empty $NotSubsetInMCS$ always returns true. $D(i, j, \{j, k\})$ returns true and $\{j, k\}$ is added to MCS. $D(i, j, \{j, l\})$ and $D(i, j, \{j, m\})$ return false. $MCS = \{\{j, k\}\}$. *Iteration 3*: Diagnosis groups which contain three elements that include fault j: $\{j, k, l\}$, $\{j, k, m\}$, $\{j, l, m\}$. Since $MCS = \{\{j, k\}\}$ $NotSubsetInMCS$ returns false for $\{j, k, l\}$ and $\{j, k, m\}$. $D(i, j, \{j, l, m\})$ returns true and $\{j, l, m\}$ is added to MCS. $MCS = \{\{j, k\}, \{j, l, m\}\}$. *Iteration 4*: Diagnosis groups containing four elements that include fault j: $\{j, k, l, m\}$. Since $MCS = \{\{j, k\}, \{j, l, m\}\}$ $NotSubsetInMCS$ returns false for $\{j, k, l, m\}$. $MCS = \{\{j, k\}, \{j, l, m\}\}$.

The algorithm is suitable for systems in which faults are resolved after they are diagnosed and future occurrences of the fault have no further impact on the system. Repeating faults can be analysed by substituting Formula 2 with a formula that reasons about the diagnosability of a fault after each injection of the fault. A more dynamic analysis would consider the case where one agent may diagnose faults if another agent is not available for diagnosis, i.e., the disjunction of agents diagnosing the disjunction of faults. Another case to consider is the conjunction of faults, for situations in which different faults injected into the system are diagnosed as a single fault. Other diagnosability specification patterns from [12] can also be applied to reason about possible diagnosis, group diagnosis, and the propagation of the knowledge of faults through the system. These extensions seem feasible but would require a different definition of the cut set function and minimal cut set restriction.

The implementation of the diagnosability analysis algorithm requires an interface to a temporal-epistemic logic model checker, the definitions of faults and agents, and the mutated interpreted system model. In particular, function D must use the model checker to verify specifications on the mutated model. We describe how these requirements are met by a fault analysis module interface which is defined as part of a framework for implementing fault analysis algorithms as modules in the next section.

4 A Framework for Integrating Fault Analysis Modules

As part of a practical approach towards verifying fault tolerant MAS we constructed a framework for integrating fault analysis modules with the model checker MCMAS and a compiler for injecting automatically faults into a MCMAS program [12]. Doing so provides a manner in which powerful fault analysis

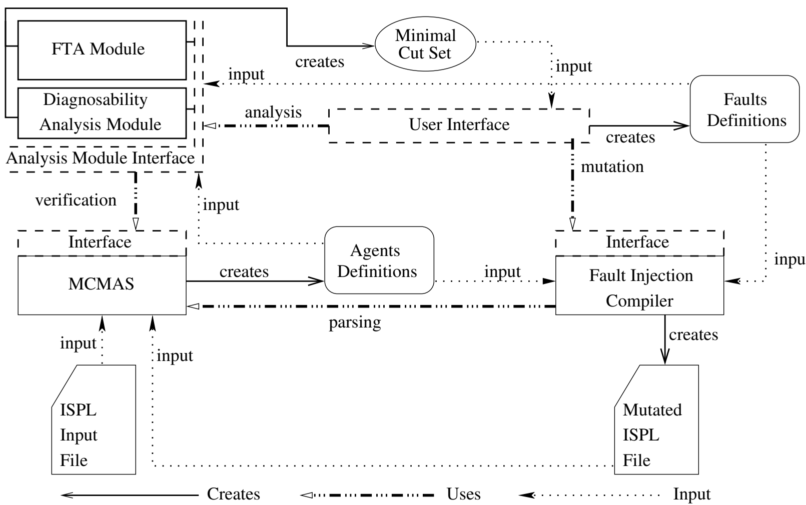


Fig. 2. Integrating fault analysis modules using the analysis module interface

modules based on algorithms such as the one presented in Section 3 can be implemented. The high level of automation achieved by fault analysis modules makes them particularly usable for non-experts in verification who are working on the design of fault tolerant MAS.

We outline the framework in Figure 2. Any analysis module can be inserted into the framework via the *analysis module interface* which provides the module with an interface from which it is invoked by the user, the definitions of the agents and faults which can be used to construct formulas to be used for analysis, an interface to MCMAS to verify formulas on a mutated ISPL program, and a method to display to the user a minimal cut set in a graphical or report format. We further describe the framework as follows.

User interaction: the user interface allows the user to define the faults which are to be injected into an ISPL program; save a file containing the mutated ISPL program which includes the injected faults; and analyse automatically the behaviour of the model defined by the mutated ISPL program by using any available fault analysis module. The process of defining faults includes their name, mutation rules, timing options, etc. If automatic analysis of faults is required then the compiler interface is used to create a file containing the mutated ISPL program for analysis, and the fault analysis module selected by the user is initiated though the analysis module interface to create a minimal cut set which is passed back from the analysis module interface to be displayed in either a graphical or report format.

MCMAS: the MCMAS interface is used to perform the parsing of an ISPL file and perform the verification of specifications against a mutated ISPL program. The parsing procedure results in the creation of agents definitions including

their name, protocol, transition relation, actions, etc. Passing a formula to the MCMAS interface instructs MCMAS to perform verification of the specification against the mutated ISPL program and the result of the verification is passed back through the MCMAS interface.

Fault injection compiler: the compiler interface is used to request the mutation of an ISPL program. MCMAS is used to parse the file in which the ISPL program containing correct behaviour is defined in order to create the agents definitions. The agents definitions and faults definitions are utilised by the compiler to save a file consisting of the mutated ISPL program which contains both correct and faulty behaviour including the injected faults.

Automatic fault analysis: the analysis module interface is used to invoke any available fault analysis module to perform analysis on a mutated ISPL program that has been created using the agents definitions and faults definitions. The module identifies a minimal cut set as a result of the analysis. The agents definitions and faults definitions are utilised to construct the formulas which are used for the analysis. MCMAS is used to verify these specifications against the mutated ISPL program. The minimal cut set identified as a result of the analysis is passed back through the interface so that it can be displayed to the user.

The framework is implemented in C++ (for linux operating systems) so that modules can be easily integrated into the framework in an object oriented manner. We implemented into the framework an FTA module which displays fault trees in a graphical format, and a diagnosability analysis module based on the algorithm in Section 3 which displays a report on the diagnosability of faults by agents in the system. The framework and these modules are included as part of the fault injection compiler which is available for public use [11].

5 Automatic Diagnosability Analysis of the IEEE 802.5 Token Ring Protocol

The IEEE 802.5 token ring protocol is a widely used local area network (LAN) protocol in which network nodes are logically organised in a ring. The data circulates in one direction around the ring via a token passed from node to node. The network is logically defined as a ring topology and physically defined as a star topology. Fault tolerance is achieved by physically disconnecting faulty nodes and re-establishing the logical ring to bypass them. The protocol employs a distributed diagnosis mechanism to diagnose faulty nodes for disconnection.

Tokens containing fault information are sent around the ring when a fault occurs, which allows nodes to diagnose faults. One node on the ring is designated as an *active monitor* which diagnoses and resolves *soft faults*, i.e., faults that can be resolved without requiring a node to be disconnected. The other nodes are designated as *standby monitors* which diagnose *hard faults* on themselves and their nearest upstream neighbour i.e., faults that are resolved by disconnecting and bypassing nodes. Once a fault is diagnosed, information can be propagated around the ring to inform nodes that a fault has been diagnosed.

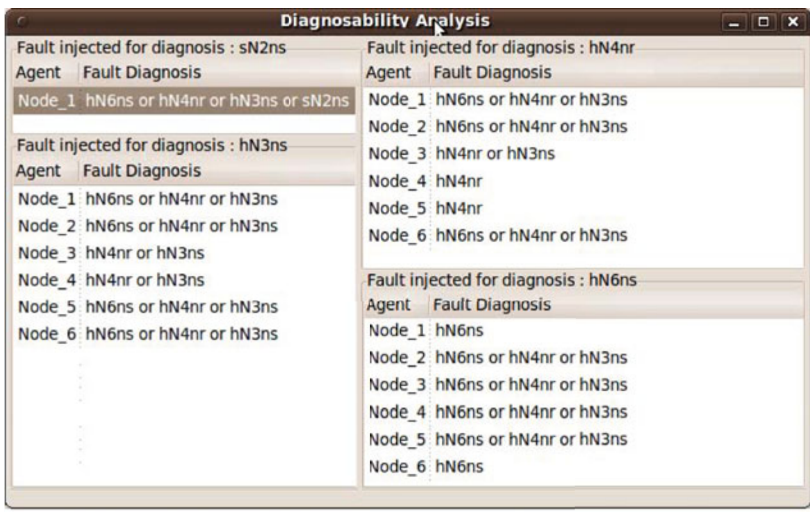


Fig. 3. A diagnosability analysis report on the token ring model

To illustrate the practical application of our diagnosability analysis module we use the implementation of the token ring protocol in ISPL from [12], which we refer the reader to for in-depth details of the implementation and the faults we injected into the model. The model contains 6 nodes labelled $N1, \ldots, N6$ with Node 1 designated as the active monitor; the token circulates clockwise from $N1$ to $N6$. Several different faults were chosen to be injected into different nodes of the model so that the diagnosability of the active and standby monitors could be analysed. Once a fault has been resolved it has no further impact on the protocol. We used the *variable value replace* and *stuck at select* mutation rules combined with various timing options to define the following faults for injection:

$sN2ns$: node 2 stops sending tokens (soft fault).
$hN3ns$: node 3 stops sending tokens (hard fault).
$hN4nr$: node 4 stops receiving tokens (hard fault).
$hN6ns$: node 6 stops sending tokens (hard fault).

Once the faults had been defined for injection, automatic diagnosability analysis was selected to be performed on the mutated token ring model containing correct and faulty behaviour. The diagnosability analysis took approximately 22 minutes to complete on an Intel Pentium 2.5GHz Dual Core E5200 processor using approximately 57MB of memory. The number of reachable states was approximately 2.3×10^5 out of a possible 1.4×10^{13}.

The diagnosability report displayed to the user which shows the results from the analysis is illustrated in Figure 3. The report displays the minimal cut sets identified for each injected fault and agent combination. For each injected fault a list is shown of the most specific diagnosis of that fault (under the heading *Fault Diagnosis*) that each agent can make.

The report allows us to determine a number of diagnosability properties of the token ring protocol as follows; Firstly, when the soft fault sN2ns is injected it is diagnosed by the active monitor as a possible soft or hard fault, thus, the active monitor uses the same mechanism to diagnose all faults; Secondly, only the active monitor can diagnose a soft fault; Thirdly, when a hard fault is injected, all monitors can diagnose the occurrence of a hard fault that has affected themselves or their nearest upstream neighbour; Finally, all monitors can diagnose the occurrence of a hard fault on the ring.

The diagnosability properties we described are critical to achieving fault tolerance in the token ring protocol. Soft faults need to be diagnosed by the active monitor for resolution, hard faults are resolved by the standby monitor if the fault has occurred on itself or its nearest upstream neighbours, and all other monitors need to know that a hard fault has occurred so that they can propagate information about the fault around the ring.

Without having to write any specifications, and in the absence of reasoning about fault resolution mechanisms, we have demonstrated that the token ring protocol accurately diagnoses faults. This illustrates how automatic diagnosability analysis can be usefully applied during the early design stages of fault tolerant MAS.

6 Related Work

The majority of the previous work on combining fault injection with model checking [2, 3, 4, 5, 17] is limited to temporal logic model checking. Moreover, the approaches are primarily concerned with verifying properties of fault tolerance and do not analyse diagnosability. A platform in which analysis artifacts including an FTA module are integrated with an automatic fault injection tool and a temporal logic model checker is described in [4]. The general approach to producing fault analysis artifacts in our paper is similar to the implementation of the automatic FTA algorithm in [3].

The earlier work on combining fault injection with temporal-epistemic model checking [12, 13] implements a compiler for injecting faults automatically into an ISPL model, and defines a number of formulas for verifying fault tolerance, recoverability, and diagnosability. Specifications are hand written to analyse diagnosability and automatic diagnosability analysis is not considered.

The previous work on analysing diagnosability [7, 9, 22, 24] considers discrete event systems [22, 24], and model based diagnosis systems [7, 9]. The main focus of the work in [9, 22, 24] is the formalisation of the diagnosability problem and less attention is given to the practicality of the proposed algorithms for analysis. A practical approach to verifying diagnosability using temporal logic model checking is given in [7]. In this approach, a coupled twin model of the diagnosis system must be constructed so that diagnosability can be expressed as a temporal specification. This implies that the modelling component of the technique is significantly more difficult in comparison to injecting automatically faulty behaviour. The practicality of this approach is not examined for distributed systems.

7 Conclusions

In this paper we presented a methodology for automatic diagnosability analysis based on fault injection and temporal-epistemic model checking. We implemented an algorithm to analyse automatically the diagnosability of faults by agents in a system. As part of the implementation we defined a framework for integrating fault analysis modules with the model checker MCMAS and a fault injection compiler. We demonstrated the practical usefulness of our approach by analysing automatically diagnosability in a model of the token ring protocol which utilises distributed diagnosis to achieve fault tolerance.

We regard our methodology as an important step in the development of practical tools for verifying fault tolerant MAS. Our framework can be used to build powerful automatic fault analysis modules which are user friendly. These are particularly useful for non-experts in verification who are working on the design of fault tolerant MAS. The analysis can be employed at an early stage of design with a high level of automation. These aspects of our work encourage a unified design and verification approach for fault tolerant MAS.

In future work we intend to implement diagnosability analysis modules that analyse group diagnosis, and the propagation of the knowledge of faults through the system. Our analysis modules will be applied to autonomous vehicle control systems and we aim to provide a powerful analysis tool for engineers working on the design of these systems.

Acknowledgement

The research described in this paper is partly supported by EPSRC funded project EP/E02727X/1.

References

[1] Avizienis, A., Laprie, J.-C., Randell, B., Landwehr, C.E.: Basic concepts and taxonomy of dependable and secure computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
[2] Bernardeschi, C., Fantechi, A., Gnesi, S.: Model checking fault tolerant systems. Software Testing, Verification and Reliability 12(4), 251–275 (2002)
[3] Bozzano, M., Villafiorita, A.: Integrating fault tree analysis with event ordering information. In: Proceedings of ESREL 2003, pp. 247–254. Swets & Zeitlinger, Lisse (2003)
[4] Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA safety analysis platform. Software Tools for Technology Transfer 9(1), 5–24 (2007)
[5] Bruns, G., Sutherland, I.: Model checking and fault tolerance. In: Johnson, M. (ed.) AMAST 1997. LNCS, vol. 1349, pp. 45–59. Springer, Heidelberg (1997)
[6] Caspi, P., Mazuet, C., Paligot, N.R.: About the design of distributed control systems: The quasi-synchronous approach. In: Voges, U. (ed.) SAFECOMP 2001. LNCS, vol. 2187, pp. 215–226. Springer, Heidelberg (2001)

[7] Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: Proceedings of IJCAI 2003, pp. 363–369. Morgan Kaufmann, San Francisco (2003)
[8] Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
[9] Console, L., Picardi, C., Ribaudo, M.: Diagnosis and diagnosability analysis using PEPA. In: Proceedings of ECAI 2000, pp. 131–135. IOS Press, Amsterdam (2000)
[10] Cristian, F.: Understanding fault-tolerant distributed systems. Commun. ACM 34(2), 56–78 (1991)
[11] Ezekiel, J., Lomuscio, A.: MCMAS fault injection compiler project page, http://www.doc.ic.ac.uk/~jezekiel/ficompiler.html
[12] Ezekiel, J., Lomuscio, A.: An automated approach to verifying diagnosability in multi-agent systems. In: Proceedings of SEFM 2009, pp. 51–60. IEEE, Los Alamitos (2009)
[13] Ezekiel, J., Lomuscio, A.: Combining fault injection and model checking to verify fault tolerance in multi-agent systems. In: Proceedings of AAMAS 2009, pp. 113–120. IFAAMAS (2009)
[14] Fagin, R., Halpern, J.Y., Vardi, M.Y., Moses, Y.: Reasoning about knowledge. MIT Press, Cambridge (1995)
[15] Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 479–483. Springer, Heidelberg (2004)
[16] Iyer, R.: Experimental evaluation. In: Proceedings of FTCS-25, pp. 115–132. IEEE, Los Alamitos (1995)
[17] 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)
[18] Kalech, M., Kaminka, G.A.: On the design of social diagnosis algorithms for multi-agent teams. In: Proceedings of IJCAI 2003, pp. 370–375. Morgan Kaufmann, San Francisco (2003)
[19] Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
[20] Mannor, S., Shamma, J.S.: Multi-agent learning for engineers. Artificial Intelligence 171(7), 417–422 (2007)
[21] Niewiadomski, A., Penczek, W., Szreter, M.: Verics 2004: A model checker for real time and multi-agent systems. In: Proceedings of CS&P 2004, Informatik-Berichte, pp. 88–99 (2004)
[22] Sampath, M., Sengupta, R., Lafortune, S., Sinnamohideen, K., Teneketzis, D.: Diagnosability of discrete-event systems. IEEE Transactions on Automatic Control 40(9), 1555–1575 (1995)
[23] Vesley, W., Goldberg, F., Roberts, N., Haasl, D.: Fault tree handbook. Technical Report NUREG-0492, Systems and Reliability Research Office of Nuclear Regulatory Research U.S. Nuclear Regulatory Comission (1981)
[24] Wang, Y., Yoo, T.-S., Lafortune, S.: Diagnosis of discrete event systems using decentralized architectures. Discrete Event Dynamic Systems 17(2), 233–263 (2007)
[25] Wooldridge, M.J.: Reasoning about Rational Agents. MIT Press, Cambridge (2000)

Making the Right Cut in Model Checking Data-Intensive Timed Systems

Rüdiger Ehlers, Michael Gerke, and Hans-Jörg Peter

Reactive Systems Group
Saarland University
66123 Saarbrücken, Germany
{ehlers,gerke,peter}@cs.uni-saarland.de

Abstract. The success of industrial-scale model checkers such as UPPAAL [3] or NuSMV [12] relies on the efficiency of their respective symbolic state space representations. While difference bound matrices (DBMs) are effective for representing sets of clock values, binary decision diagrams (BDDs) can efficiently represent huge discrete state sets. In this paper, we introduce a simple general framework for combining both data structures, enabling a joint symbolic representation of the timed state sets in the reachability fixed point construction. In contrast to other approaches, our technique is robust against intricate interdependencies between clock constraints and the location information. Especially in the analysis of models with only few clocks, large constants, and a huge discrete state space (such as, e.g., data-intensive communication protocols), our technique turns out to be highly effective. Additionally, our framework allows to employ existing highly-optimized implementations for DBMs and BDDs without modifications. Using a prototype implementation, we are able to verify a central correctness property of the physical layer protocol of the FlexRay communication protocol [15] taking an unreliable physical layer into account.

1 Introduction

The verification of asynchronous protocols or *globally asynchronous locally synchronous* (GALS) hardware is a challenging task as it often requires dealing with intricate timing dependencies and huge state spaces at the same time. Manual correctness proofs are relatively hard to perform as special timing cases can be overlooked more easily than in the purely synchronous setting. Consequently, working with automated correctness provers, in particular by performing model checking, is the predominant approach carried out in this field. For keeping track of the timing correctly, a rich theory of timed automata [1] has been developed, which forms the theoretical foundation for model checking tools such as UPPAAL [3], KRONOS [23], or RED [21].

State-of-the-art model checking approaches for timed systems can broadly be classified into two categories: *semi-symbolic* approaches and *fully symbolic* approaches [19]. In semi-symbolic approaches, on the one hand, the discrete part of

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 565–580, 2010.
© Springer-Verlag Berlin Heidelberg 2010

the system under consideration is represented explicitly while clock valuations are lumped together into clock zones. These techniques are well-suited for systems with a small discrete state space. Fully symbolic approaches, on the other hand, represent *both* parts of the system in a symbolic way to lower the effect of state space explosion. For settings with a predominant control structure (such as, e.g., data-intensive communication protocols), this is broadly considered to be the more promising way to go as the discrete state space is often too large to be represented explicitly even with modern computers.

Fully symbolic timed model checking is a challenging problem that attracted a lot of interest during the last decade [9,16,4,21,6,19,22]. Some approaches rely on restricting the type of timing of the system in a way that it can be discretized more efficiently [6] or approximating the precise clock values to discrete time steps [9]. In both cases, reduced ordered binary decision diagrams (BDDs) [10,11] have been used as the uniform data structure for both time and the discrete part of the system. For such settings, it has been observed that the BDDs can blow-up significantly due to interdependencies in the timing behavior of the system [9], rendering this approach problematic.

Basically, this leaves us with two ways of solving this problem. The first one builds on using a different symbolic data structure like and-inverter graphs [17] or conjunctive normal form clause sets, thus avoiding this blow-up. Here, canonicity of the representation is renounced, rendering the application of the basic reachability fixed point algorithm difficult. The second way is based on keeping the data structures for clock zones and discrete state sets separate. One promising approach in this direction is to combine difference bound matrices (DBMs) [13] and BDDs. So far, to the best of our knowledge, this combination has only been used for *approximating* the set of reachable states [22].

In this paper, we recall the idea of decoupling the clock zone from the discrete state set representation. We use sets of pairs of DBMs and BDDs to represent reachable states in the classical fixed point computation. A timed state is contained in a state set if the set contains a DBM/BDD pair such that the state's clock valuation satisfies the DBM and the discrete part satisfies the BDD. To the best of our knowledge, this is the first application of this data structure with the standard fixed point algorithm.

Especially in the verification of asynchronous communication protocols, where the timing behavior is usually complicated but the number of distinct arising clock zones is small, our approach turns out to be very efficient since we benefit from the well-suitedness of DBMs for representing intricate timing constraints without cluttering the BDDs with timing dependencies. As the new approach allows the usage of the standard fixed point construction for finding the set of reachable states, it is simpler than the aforementioned techniques. The drawback of having some timed state potentially being present in more than one such pair is easily compensated by the additional possibility to perform a mixed breadth-first and depth-first search in this setting: by preferring discrete steps, we can forward the progress in exploring the discrete state space of the system in one

clock zone to successor clock zones, leading to faster termination of the model checking process.

As a proof of concept, we demonstrate the applicability of our technique by verifying the physical layer protocol of the FlexRay communication protocol [15] which follows the GALS paradigm: the setting can be described using only two clocks, each modeling the local pulsing in the discrete circuits of the sender and the receiver. Our approach is thus ideally suited for this interesting verification task as it exploits the model's restricted timing behavior, which enables the use of BDDs to tackle the huge arising discrete state space.

In Sect. 2, we begin our presentation with some preliminary definitions. Section 3 then describes the general approach, followed in Sect. 4 by an explanation of how example traces certifying the satisfaction of a reachability property are generated. Afterwards, we describe our FlexRay model which is then used for an experimental evaluation of the approach in Sect. 5. Finally, Sect. 6 concludes with an outlook on the possible evolution of this approach.

Related work. Developing fully symbolic timed state space representations has been an active field of research in the last decade. Møller et al. introduced *difference decision diagrams* (DDDs) [16], a BDD-like data structure in which each diagram node is labeled with a difference constraint. Here, the Boolean constraints, represented as special differences, are interleaved with the clock constraints in the diagram structure. Unfortunately, there is no implementation of their prototype model checker available. Based on DDDs, Behrmann et al. proposed *clock difference diagrams* (CDDs) [4] featuring deterministic interval-based branching at each node level. However, a combination of CDDs with BDDs enabling a fully symbolic state space exploration was only briefly discussed but has not been thoroughly investigated yet. As a further extension, Wang proposed *clock restriction diagrams* (CRDs) [21] where the branching decision depends on overlapping upper bounds and unrestricted constraints are omitted. Experiments with the CRD-based model checker RED suggest that the approach works well on standard timed automata benchmarks having many clocks and causing only a moderate discrete blow-up. However, in our experiments, RED runs out of memory on the FlexRay case study.

Based on *closed timed automata*, a restricted form of classical timed automata where only nonstrict clock constraints are allowed, Beyer introduced an integer semantics where clock values and location configurations can be represented jointly in a single BDD [6]. Similarly, Bozga et al. approximated the precise clock values to discrete time steps, also resulting in a pure discrete semantics allowing a state space representation using a single BDD [9]. Besides the loss of expressivity in the modeling of timed systems, in both approaches, it has been observed that the BDDs can blow-up significantly due to interdependencies in the timing behavior of the system.

Seshia and Bryant solved the TCTL model checking problem by representing sets of states by difference logic formulas which are, in turn, represented as BDDs using a binary encoding [19]. The clock differences that need to be tracked in the fixed-point computation are encoded in so-called transitivity constraints,

which are added on-the-fly during the model checking process. Even though they added some specialized optimizations for this case, the experimental results are inconclusive.

The idea of combining DBMs and BDDs was independently developed by Yamane and Nakamura [22] for implementing an *abstraction* technique proposed by Dill and Wong-Toi [14]. Our approach, however, uses DBM/BDD combinations for a fully symbolic state space representation in the *precise* computation of the reachable states of a timed system.

Previous correctness proofs of the physical layer protocol of the FlexRay communication protocol [15] were obtained in a deductive way. In this line of research, a fully reliable physical layer without any bit flips is assumed [7]. Our correctness proof, which is obtained via model checking, bases upon a more realistic setting taking an unreliable physical layer into account.

2 Preliminaries

2.1 Timed Systems

Timed Automata. The components of a timed system are represented by *timed automata*. A timed automaton [1] is a tuple $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$, where L is a finite set of (control) locations, $l_0 \in L$ is the initial location, $I : L \to \mathcal{C}(X)$ maps each location to an invariant, Σ is a finite set of actions, $\Delta \subseteq (L \times \Sigma \times \mathcal{C}(X) \times 2^X \times L)$ is a transition relation, X is a finite set of real valued clocks, and $\mathcal{C}(X)$ is the set of clock constraints over X. A clock constraint $\varphi \in \mathcal{C}(X)$ is of the form

$$\varphi = \mathbf{true} \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2,$$

where x is a clock in X and c is a constant in $\mathbb{N}_0$. We say that a timed automaton is invariant-free if $I(l) = \mathbf{true}$ for all locations $l \in L$. A *clock valuation* $\boldsymbol{t} : X \to \mathbb{R}_{\geq 0}$ assigns a nonnegative value to each clock and can also be represented by a $|X|$-dimensional vector $\boldsymbol{t} \in \mathcal{R}$, where $\mathcal{R} = \mathbb{R}_{\geq 0}^X$ denotes the set of all clock valuations.

The states of a timed automaton are pairs $(l, \boldsymbol{t})$ of locations and clock valuations. Timed automata have two types of transitions: *timed transitions*, where only time passes and the location remains unchanged, and *discrete transitions*. In a timed transition, denoted by $(l, \boldsymbol{t}) \xrightarrow{a} (l, \boldsymbol{t} + a \cdot \mathbf{1})$, the same nonnegative value $a \in \mathbb{R}_{\geq 0}$ is added to all clocks such that, for each $0 \leq d \leq a$, $\boldsymbol{t} + d$ satisfies the location invariant $I(l)$. A discrete transition, denoted by $(l, \boldsymbol{t}) \xrightarrow{a} (l', \boldsymbol{t}')$ for some $a \in \Sigma$, is a transition $\delta = \langle l, a, \varphi, \lambda, l' \rangle$ of Δ such that $\boldsymbol{t}$ satisfies the clock constraint φ of δ, and $\boldsymbol{t}' = \boldsymbol{t}[\lambda := 0]$ is obtained from $\boldsymbol{t}$ by setting the clocks in λ to 0 and satisfies the location invariant $I(l')$.

We say that a finite sequence $a_1 \ldots a_n \in (\Sigma \cup \mathbb{R}_{\geq 0})^*$ of transitions is in the *language of* $\mathcal{A}$ ($a_1 \ldots a_n \in \mathcal{L}(\mathcal{A})$) if there is a path $s_0 \xrightarrow{a_1} s_1 \ldots s_{n-1} \xrightarrow{a_n} s_n$ such that for all $1 \leq i \leq n$, the individual $s_i = (l_i, \boldsymbol{t}_i)$ are states of the automaton, s_0 is an initial state (that is, l_0 is the initial location and $\boldsymbol{t}_0 = \mathbf{0}$ is the zero vector),

and $s_{i-1} \xrightarrow{a_i} s_i$ are transitions of $\mathcal{A}$. We write $s_0 \longrightarrow^* s_n$ for the existence of a finite sequence $a_1 \dots a_n \in (\Sigma \cup \mathbb{R}_{\geq 0})^*$ of transitions with $s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \dots \xrightarrow{a_n} s_n$, and call a state s *reachable* iff there is an initial state s_0 with $s_0 \longrightarrow^* s$.

Composition. Timed automata can be composed to networks, in which the automata run in parallel and synchronize on shared actions. For two timed automata $\mathcal{A} = (L_1, l_0^1, I_1, \Sigma_1, \Delta_1, X_1)$ and $\mathcal{A}' = (L_2, l_0^2, I_2, \Sigma_2, \Delta_2, X_2)$ with disjoint clock sets $X_1 \cap X_2 = \emptyset$, the *parallel composition* $\mathcal{A}_1 \| \mathcal{A}_2$ is the timed automaton $(L_1 \times L_2, (l_0^1, l_0^2), I, \Sigma_1 \cup \Sigma_2, \Delta, X_1 \cup X_2)$, where $I(l_1, l_2) = I_1(l_1) \wedge I_2(l_2)$ for all $l_1 \in L_1$ and $l_2 \in L_2$, and Δ is the smallest set that contains

- for $a \in \Sigma_1 \cap \Sigma_2$, $\langle (l_1, l_2), a, \varphi_1 \wedge \varphi_2, \lambda_1 \cup \lambda_2, (l_1', l_2') \rangle$ if $\langle l_1, a, \varphi_1, \lambda_1, l_1' \rangle \in \Delta_1$ and $\langle l_2, a, \varphi_2, \lambda_2, l_2' \rangle \in \Delta_2$,
- for $a \in \Sigma_1 \setminus \Sigma_2$, $\langle (l_1, l_2), a, \varphi_1, \lambda_1, (l_1', l_2) \rangle$ if $\langle l_1, a, \varphi_1, \lambda_1, l_1' \rangle \in \Delta_1$, and
- for $a \in \Sigma_2 \setminus \Sigma_1$, $\langle (l_1, l_2), a, \varphi_2, \lambda_2, (l_1, l_2') \rangle$ if $\langle l_2, a, \varphi_2, \lambda_2, l_2' \rangle \in \Delta_2$.

In the following, we only consider the *global timed automaton* that is obtained from the composition of the system's component automata. Note that control-related concepts such as synchronization, parallel composition, or integer variables are just technicalities in the construction of the symbolic discrete transition relation; they do not have to be considered in the actual model checking procedure.

Finite Semantics. The decidability of the reachability problem of timed automata relies on the existence of the *region equivalence relation* [1] on $\mathcal{R}$ which has a finite index.

For a timed automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$, we call the value of a clock $x \in X$ *maximal* if it is strictly greater than the highest constant c_{max} any clock is compared to.[1] We say that two clock valuations $\boldsymbol{t}_1, \boldsymbol{t}_2 \in \mathcal{R}$ are in the same *clock region*, denoted $\boldsymbol{t}_1 \sim_R \boldsymbol{t}_2$, if

- the set of clocks with maximal value is the same in $\boldsymbol{t}_1$ and in $\boldsymbol{t}_2$ ($\forall x \in X : \boldsymbol{t}_1(x) > c_{max} \Leftrightarrow \boldsymbol{t}_2(x) > c_{max}$), and
- $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ agree (1) on the integer parts of the clock values, (2) on the relative order of the noninteger parts of the clock values, and (3) on the equality of the noninteger parts of the clock values with 0. That is, for all clocks x and y with nonmaximal value, it holds that (1) $\lfloor \boldsymbol{t}_1(x) \rfloor = \lfloor \boldsymbol{t}_2(x) \rfloor$, (2) $\widehat{\boldsymbol{t}}_1(x) \leq \widehat{\boldsymbol{t}}_1(y) \Leftrightarrow \widehat{\boldsymbol{t}}_2(x) \leq \widehat{\boldsymbol{t}}_2(y)$, and (3) $\widehat{\boldsymbol{t}}_1(x) = 0$ if, and only if, $\widehat{\boldsymbol{t}}_2(x) = 0$, where $\widehat{\boldsymbol{t}}_i(x) = \boldsymbol{t}_i(x) - \lfloor \boldsymbol{t}_i(x) \rfloor$ for $i \in \{1, 2\}$.

We denote with $[\boldsymbol{t}]_R = \{\boldsymbol{t}' \in \mathcal{R} \mid \boldsymbol{t} \sim_R \boldsymbol{t}'\}$ the clock region $\boldsymbol{t}$ belongs to. We say that two states $s_1 = (l_1, \boldsymbol{t}_1)$ and $s_2 = (l_2, \boldsymbol{t}_2)$ of $\mathcal{A}$ are *region-equivalent*, denoted by $s_1 \sim_R s_2$, if their locations are the same ($l_1 = l_2$) and the clock valuations are in the same clock region ($\boldsymbol{t}_1 \sim_R \boldsymbol{t}_2$), and denote with $[(l, \boldsymbol{t})]_R = \{(l, \boldsymbol{t}') \in L \times \mathcal{R} \mid \boldsymbol{t} \sim_R \boldsymbol{t}'\}$ the equivalence class of region-equivalent states that $(l, \boldsymbol{t})$ belongs to.

Regions are a suitable semantics for the abstraction of timed automata because they essentially preserve the language: if there is a discrete transition $s \xrightarrow{a} s'$ from a state s to a state s' of a timed automaton, then there is, for all

[1] c_{max} is sometimes called the clock ceiling.

states r with $r \sim_R s$, a state r' with $r' \sim_R s'$ such that $r \xrightarrow{a} r'$ is a discrete transition with the same label. For timed transitions, a slightly weaker property holds: if there is a timed transition $s \xrightarrow{t} s'$ from a state s to a state s', then there is, for all states r with $r \sim_R s$, a state r' with $r' \sim_R s'$ such that there is a timed transition $r \xrightarrow{t'} r'$ (but possibly with $t' \neq t$).

The *finite semantics* of a timed automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$ is the finite graph $[\![\mathcal{A}]\!] = (S, s_0, T)$ where

- the symbolic state set $S = \{[(l, \boldsymbol{t})]_R \mid (l, \boldsymbol{t}) \in L \times \mathcal{R}\}$ of $[\![\mathcal{A}]\!]$ is the set of equivalence classes of region-equivalent states of $\mathcal{A}$, with
- the initial state $s_0 = [(l_0, \boldsymbol{t}_0)]_R$, and
- the set $T = \{(s, s') \in S \times S \mid \exists r \in s,\ r' \in s', a \in \Sigma \cup \mathbb{R}_{\geq 0}. r \xrightarrow{a} r'\}$ of transitions.

The finite semantics is reachability-preserving:

Lemma 1. *[1] For a timed automaton $\mathcal{A} = (L, l_0, I, \Sigma, \Delta, X)$ there is a finite path from a state $(l, \boldsymbol{t})$ to a state $(l', \boldsymbol{t}')$ if, and only if, there is a finite path from $\big[(l, \boldsymbol{t})\big]_R$ to $\big[(l', \boldsymbol{t}')\big]_R$ in $[\![\mathcal{A}]\!]$.*

Clock Zones. A coarser finite representation of $\mathcal{R}$ can be obtained by considering *clock zones*. A clock zone z is represented by a conjunction of clock difference constraints of the form $x - y \prec_{x,y} c_{x,y}$, where $x, y \in X \cup \{x_0\}$, for an $x_0 \notin X$, $\prec_{x,y} \in \{\leq, <\}$, and $c_{x,y} \in \mathbb{Z} \cup \{\infty\}$. A clock valuation $\boldsymbol{t}$ satisfies z, written as $\boldsymbol{t} \in z$, if $\boldsymbol{t}' = \boldsymbol{t} \cup \{x_0 \mapsto 0\}$ satisfies each constraint $x - y \prec_{x,y} c_{x,y}$ from z: $\boldsymbol{t}'(x) - \boldsymbol{t}'(y) \prec_{x,y} c_{x,y}$. We define $\mathcal{Z}$ as the set of all clock zones.

A data structure for representing clock zones are difference bound matrices (DBMs) [13], which allow, for two clock zones $z, z' \in \mathcal{Z}$, an efficient implementation of the operations (1) intersection $z \wedge z'$, (2) clock reset $z[\lambda := 0]$, and (3) elapsing of time $z^{\Uparrow}$ (see [5] for an overview). Note that, in order to ensure termination of the forward analysis, we implicitly apply a maximal constant extrapolation after executing a time elapse. We denote z_0 as the clock zone that only comprises the initial clock valuation $\mathbf{0}$.

2.2 Binary Decision Diagrams

For representing sets of locations symbolically we use *reduced ordered binary decision diagrams* (BDDs) [10,11], which represent functions $f : 2^V \rightarrow \mathbb{B}$ for some finite set of variables V. Since they are well-established in the context of formal verification, we do not describe their details here but rather treat them on an abstract level and only state the important operations (see [11] for an overview). Given two BDDs (or more generally, two binary functions, abbreviated as BF) f and f', we define their conjunction and disjunction as $(f \wedge f')(x) = f(x) \wedge f'(x)$ and $(f \vee f')(x) = f(x) \vee f'(x)$ for all $x \subseteq V$. The negation of a BF is defined similarly. Given some set of variables $V' \subseteq V$ and a BF f, we define $\exists V'.f$ as the function that maps all $x \subseteq V$ to **true** for which there exists some $x' \subseteq V'$ such that $f(x' \cup (x \setminus V')) = \textbf{true}$.

Algorithm 1. Least fixed point construction for computing the set of reachable states R.

```
R_0 := {initial states}
i := 0
repeat
  i := i + 1
  R_i := R_{i-1} ∪ post(R_{i-1})
until R_i = R_{i-1}
R := R_i
```

2.3 Reachability Model Checking

Model checking reachability properties is carried out by computing the set of reachable states and testing whether some goal state is contained in this set. In this paper, w.l.o.g., we only consider properties of the form $\exists\Diamond\phi$, that is, "is there an execution of the system such that ϕ is eventually reached", where ϕ is a Boolean state predicate defining the goal states. The classical fixed point construction for this task is given in Algorithm 1. It relies on the existence of an efficiently computable post operator for computing all successor states of a given set of states.

When using BDDs for storing state sets in this algorithm, usually it is beneficial to pre-compute a Boolean encoding of the *transition relation* of the system for usage in the post operator. It contains precisely the pairs of states (s, s') for which there exists a transition from s to s'. For a comprehensive overview of building such a relation and using it in the post operator, see [2].

3 Fully Symbolic Real-Time Model Checking

In this section, we present the basic building blocks of our approach, namely the timed state set representation that is used and how the basic fixed point algorithm for computing the set of reachable states can be extended in order to be applicable to this representation. For a clear separation of concerns, we describe our representation in a general way and abstract from the actual choices of data structures for representing clock zones (CZ) and discrete location sets (LS). While for our actual implementation of the approach (as described in Sect. 5), DBMs and BDDs are used, the general idea is applicable to all suitable data structure types (such as, e.g., CDDs [4]). Thus, future alternatives for storing clock zones and location sets can also be used with our approach.

We start with a presentation for invariant-free timed automata and demonstrate the application of our algorithm on an example network. We then show how to extend the idea to include support for invariants. The section closes with some remarks on optimizations to the algorithm.

The starting point for our approach are sets of CZ/LS pairs which permit representing the timed and discrete parts of sets of states separately. That means,

sets of states $\mathcal{S}$ in the fixed point computations are defined as partial functions $\mathcal{S} : \mathcal{Z} \rightharpoonup 2^L$. For such a so-called *clock zone map* (CZM), a state $(l, \boldsymbol{t})$ is contained in S if for some $z \in \mathcal{Z}$, $\boldsymbol{t} \in z$ and $l \in \mathcal{S}(z)$. Note that we do not require the choice of z to be unique.

3.1 Computing the Reachable States Using CZMs

In the following, we describe how to adapt the basic fixed point construction for computing the set of reachable states given in Algorithm 1 to work with CZMs. The first step is to partition the overall transition relation of the system: for each combination of clock guards and resets that occurs along some transition, we build a separate transition relation containing all transitions corresponding to the guard/reset pair. This step separates timing concerns from the discrete transitions of the system and makes it easy to compute successor clock zones from a given source clock zone and some guard/reset pair. Note that, when using BDDs for representing location sets, it is *not* necessary to enumerate the product locations in the global timed automaton explicitly if the system is given as a network of timed automata, as the synchronization between the components can be encoded *symbolically*.

After building the transition relations, the usual fixed point computation is performed, with the small modification of iterating over all such guard/reset pairs in every step. After each discrete transition, we also compute the set of possible timed transitions that can follow in order to obtain the successor clock zone. Algorithm 2 shows the details of these steps.

In each round, the algorithm iterates over the set of reachable states contained in the respective previous pre-fixed point (stored in R). For every clock zone in the domain of R, it computes successor locations L and clock zones z' for each guard/reset pair (φ, λ) in the transition relation. Then, we check if the new CZ/LS pair (z', L) is already contained in the pre-fixed point. If this is not the case, it is added to the next pre-fixed point R'. For a more efficient computation, we furthermore track changes in the CZM in a special waiting set W in order to avoid re-considering CZ/LS pairs which have not changed since the previous round of the algorithm. The computational burden of the added inner loop in which all guard/reset pairs are iterated over is also weakened by the fact that unlike for models checkers keeping the discrete part of the system explicit, this setting allows the computation of timed successor zones for many locations at the same time.

Since the algorithm presented is essentially equivalent to the classical reachability fixed point algorithm, its correctness is guaranteed. Indeed, for every $n \in \mathbb{N}$ and timed state $(l, \boldsymbol{t})$ that is reachable from the initial state in n discrete steps, the set R contains this state after at most n iterations of computing the pre-fixed points. Note that the termination of this algorithm is also guaranteed as clock regions are never split and the number of sets of these is finite.

Algorithm 2. Computing the set of reachable states R represented as a CZM. The post operator is parametrized by the transition relation used.

1: **for all** guard/reset pairs (φ, λ) in the system **do**
2: **compute the transition relation** $T[\varphi, \lambda]$
3: **end for**
4: $R := \{z_0^{\Uparrow} \mapsto \{l_0\}\}$
5: $W := \{z_0^{\Uparrow}\}$
6: $R' := R$
7: **repeat**
8: $R := R'$
9: $W' := \emptyset$
10: **for all** $z \in W$ **do**
11: **for all** guard/reset pairs (φ, λ) **do**
12: $L := \mathsf{post}_{T[\varphi,\lambda]}(R[z])$
13: $z' := (z \wedge \varphi)[\lambda := 0]^{\Uparrow}$
14: **if** $R'[z'] \not\supseteq L$ **then**
15: $R'[z'] := R'[z'] \cup L$
16: $W' := W' \cup \{z'\}$
17: **end if**
18: **end for**
19: **end for**
20: $W := W'$
21: **until** $R = R'$

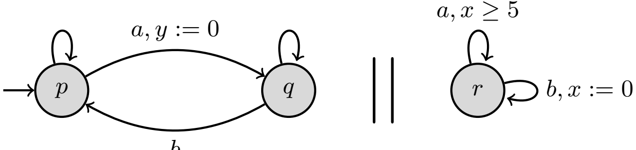


Fig. 1. An example network of timed automata

3.2 An Example

Consider the parallel composition of the timed automata depicted in Figure 1. The product automaton has two locations pr and qr, and two clocks x and y. There are three guard/reset pairs $(\mathbf{true}, \emptyset)$, $(x \geq 5, \{y\})$, and $(\mathbf{true}, \{x\})$, leading to three transition relations

- $T[\mathbf{true}, \emptyset] = \{(qr, qr), (pr, pr)\}$,
- $T[x \geq 5, \{y\}] = \{(pr, qr)\}$, and
- $T[\mathbf{true}, \{x\}] = \{(qr, pr)\}$.

When running the algorithm on this example, we initialize $R = \{(x = y = 0)^{\Uparrow} \mapsto \{pr\}\} = \{(x = y) \mapsto \{pr\}\}$ and $W = \{(x = y)\}$. Then, in the fixed point computation, we iterate over the transition relations and obtain $R = \{\{(x = y) \mapsto \{pr\}, (x \geq 5 \wedge y \leq x - 5) \mapsto \{qr\}\}$ and $W = \{(x \geq 5 \wedge y \leq x - 5)\}$. In

Algorithm 3. Replacement for lines 13–17 to the inner loop of Algorithm 2 to allow handling invariants.

```
1: for all i ∈ 𝓘 do
2:    L′ := L ∩ C(i)
3:    z′ := ((z ∧ φ)[λ := 0] ∧ i)^⇑ ∧ i
4:    if R′[z′] ⊉ L′ then
5:       R′[z′] := R′[z′] ∪ L′
6:       W′ := W′ ∪ {z′}
7:    end if
8: end for
```

the second round, we add $(x \leq y) \mapsto \{pr\}$ to R and obtain $W = \{(x \leq y)\}$. The algorithm then terminates in the following round, leaving us with $R = \big\{\{(x = y) \mapsto \{pr\}, (x \geq 5 \wedge y \leq x - 5) \mapsto \{qr\}, (x \leq y) \mapsto \{pr\}\big\}$ as the CZM representation of the set of reachable states in the system.

3.3 Handling Invariants in CZMs

To incorporate invariants in our fixed point construction, it is necessary to infer them from the location information. Here, we exploit the fact that in our models, the number of clocks and the number of distinct invariants is small. Hence, it is feasible to enumerate all possible invariants of the product timed automaton as a precomputational step.

Let $\mathcal{I}$ be the set of all invariants appearing in the product automaton of a timed system. We assume that each invariant is given in minimal form. For example, if precisely the invariants $x \leq 3$, $x \leq 4$, and $y \leq 2$ are associated to some (but not all) locations of three different components of the input network, we obtain $\mathcal{I} = \{\mathbf{true}, x \leq 3, x \leq 4, y \leq 2, x \leq 3 \wedge y \leq 2, x \leq 4 \wedge y \leq 2\}$. Furthermore, we also introduce a function $C : \mathcal{I} \to 2^L$ that maps each invariant onto the set of locations in which precisely the given invariant must hold. Note that one can easily compute $\mathcal{I}$ and C in a preprocessing step *without* constructing the product automaton.

Now, in our fixed point construction, we need to split the computed successor locations according to the mapped locations in C and compute the successor clock zones taking into account the respective invariants. Algorithm 3 contains the necessary changes to Algorithm 2. If the initial location has an active invariant, it also needs to be taken into account when computing the initial zone. Hence, we also change line 4 of Algorithm 2 to $R := \big\{z_0^{\Uparrow} \wedge I(l_0) \mapsto \{l_0\}\big\}$ and line 5 to $W := \{z_0^{\Uparrow} \wedge I(l_0)\}$.

3.4 Improving the Performance of the Approach

As an optimization of the presented technique, we propose to deviate from the strict rule of computing one pre-fixed point after the other. By storing all newly

encountered CZ/LS pairs (z, l) directly into the pre-fixed point R in the algorithm instead of R' (which is copied to R after all elements from the waiting set have been processed), computation time is saved in the case that z is in the waiting set W but not yet processed in the respective round of the fixed point computation. This can easily be seen from that fact that in such a case, the consideration of the newly reachable timed states in not delayed to the next round, resulting in a lower number of steps in total until the fixed point is reached.

Note that this also allows us to use a waiting *queue* instead of a list. Then, in line 10 of Algorithm 2, we *pop* zones (i.e., we remove every zone from W that was picked). Additionally, we modify the waiting queue such that clock zones are drawn from it prioritized by their first appearance in R. This way, the exploration of new clock zones is delayed such that the progress in computing the reachable discrete states for zones encountered earlier can be forwarded to successor zones more efficiently.

4 Guided Counter-Example Generation

The algorithm depicted so far is only capable of computing the set of reachable states. As checking if it contains some given goal state is trivial after it has been computed, this suffices to make the approach presented suitable for a typical verification task for timed systems: checking that no error state is reachable.

For cases in which some error state is reachable, however, obtaining a sequence of transitions from the initial state to some error state is desired as it helps the designer of a timed system to improve the model. Therefore, most modern model checking tools can generate such *counter-examples*. This is also possible with the improved version (see Sect. 3.4) of our fixed point construction as we explain in the following.

Suppose that our *improved* fixed point construction terminates early with a set of *forward* reachable states R comprising some error states E. Then, we execute a *nonimproved* fixed point construction to compute a sequence of pre-fixed points $F_0, \ldots, F_n$, where each F_i is restricted to R and $E_f = F_n \cap E \neq \emptyset$. Note that for all $0 \leq i \leq n$, the set F_i contains only states that are reachable after exactly i steps.

Now, we can compute a counter-example using a *backward nonimproved* fixed point construction producing a sequence of backward reachable sets of states $B_n, \ldots, B_0$ with $B_n = E_f$. For each $0 < j \leq n$, we compute B_{j-1} by picking one particular semi-symbolic state (i.e., a zone and one concrete location) from B_j, compute its predecessors, and restrict them to F_{j-1}. After each iteration j, we pick some transition connecting a state in B_{j-1} and B_j, and add it to the counter-example. Note that the computation of the predecessors can be done in a symbolic way using our technique (the adaption to the backward case is straight-forward).

5 Experimental Results

5.1 Prototype Implementation

We implemented our approach in a prototype model checker using the UPPAAL-DBM library [5] for representing DBMs and the CUDD library [20] for representing BDDs. To allow a fair comparison with UPPAAL [3] and RED [21], our tool reads automata-based specifications as input. The first step in its execution is to call the tool NOVA from the SIS toolset [18] as a back-end for finding efficient assignments of control locations to BDD variable valuations. Then, the guard/reset pairs of the given timed system are collected and, for each pair, the BDD representing the symbolic transition relation over the discrete control structure is computed (using the assignments obtained in the first step). In the last step of the preparation phase, the possible invariant combinations are collected and, for each combination, the BDD representing the associated locations is computed.

The actual fixed point computation of the reachable states is implemented as described in Sect. 3. For the state space representation, we use a hash map that maps DBMs to BDDs. We do not provide a fixed BDD variable ordering a priori or use any other insight into the model to optimize the BDD representation. Instead, our implementation only relies on the automatic on-the-fly reordering heuristics implemented in the CUDD library.

5.2 The FlexRay Communication Protocol

The emergence of drive-by-wire and the need for high bandwidths in the design of automotive electronics calls for a communication protocol that is both fast and highly reliable. The recently developed FlexRay protocol [15] represents a state-of-the-art industrial X-by-wire communication protocol that is used in many modern cars. Its purpose is to enable reliable communication between the various *electronic control units* (ECUs) that are connected by a bus. In our case study, we investigate the critical physical layer protocol of the FlexRay protocol, where a message is transmitted during a so-called *static segment* from a sending ECU to a receiving ECU. As a crucial correctness property, it is required that there is no deviation of the message received from the message sent. In the following, we explain the important details of [15] which are reflected in our model.

Clocks. Since the receiving and sending ECU are running asynchronously, we introduce two clocks to model the timing behavior. The length of a clock cycle may deviate by at most 0.15 % from the standard rate.

Bit Stream Format. The actual payload of a transmission between two ECUs has a maximal length of 262 bytes. It is embedded into a structured bit stream that consists of (1) the initial *transmission start sequence* (TSS), (2) the *frame start sequence* (FSS), (3) the individual bytes of the payload, each prepended with a *byte start sequence* (BSS), and finally (4) the *frame end sequence* (FES). Thus, the maximal bit stream length is 2638 bits.

Redundancy and Error Model. Each bit of the bit stream is fed to the bus in 8 consecutive clock cycles. As a reasonable error model, we assume that in any sequence of 5 consecutive bits on the bus, 1 bit might be flipped.

Voting. In order to compensate for the flipped bits, the receiver determines the *voted value* over the last 5 received bits (i.e., high for 3 or more high bits, low otherwise).

Strobing and Bit Clock Alignment. In order to compensate for the clock drifts, the receiver uses a counter to *strobe* the 5th out of 8 voted values. The received stream consists of the sequence of strobed values. The strobe counter is realigned at the start of the TSS or during a BSS.

5.3 Model Checking FlexRay

We modeled the physical layer protocol of the FlexRay protocol [15] as a network of timed automata[2] for usage with UPPAAL[3] [3], RED[4] [21], and our prototype model checker. As a safety property, we check the reachability of a dedicated error location which the receiver enters upon an uncompensatable deviation of the received from the sent bit stream. Table 1 shows the results of our evaluation. Unfortunately, for every payload length, RED runs out of memory (e.g., for the smallest instance it hits the 4 GB limit after 18 minutes).

The most striking observation is that our prototype overall needs much less memory than UPPAAL or RED, which allows us to verify the full payload length of 262 bytes. In fact, while our model checker's memory consumption always stays below 1 GB, UPPAAL's memory and time consumption increases linearly in the length of the payload, resulting in running out of memory with a payload length of 34 bytes or more. It is also noteworthy that in most of the cases our approach also outperforms UPPAAL w.r.t. the running time. An oscillation effect can be observed in the running times and space consumptions of our implementation which is caused by the variable reordering and caching heuristics of the CUDD library. This BDD-related phenomenon is also observable in other contexts (see, e.g., [8]). Nevertheless, the number of symbolic exploration steps increases linearly in the length of the payload and the set of encountered clock zones reaches its fixed point at a payload length of 24.

5.4 Model Checking Fischer

In addition to the FlexRay case study from Sect. 5.3, we also considered the Fischer mutual exclusion protocol, a standard benchmark from the timed model checking domain with a small discrete state space and one clock per component. Table 2 shows that the existing model checking techniques implemented in UPPAAL and RED perform better than our prototype on this benchmark.

---

[2] The models are available at `http://www.avacs.org/Benchmarks/Open/flexray.tgz`

[3] Version 4.0.11, running with aggressive space optimization – option -S2.

[4] Version 8.100511.

Table 1. Comparison of our prototype with UPPAAL on the FlexRay physical layer protocol case study. The first column shows the length of the payload in bytes. The second column states the obtained verification result. The next four columns show the number of symbolic steps (i.e., applications of the post operator) until the reachability fixed point is reached, the number of distinct clock zones encountered, the running time, and the memory consumption of our prototype model checker. The last two columns show the running time and space consumption of UPPAAL. All benchmarks were executed on an AMD Opteron processor with 2.6 GHz and 4 GB RAM.

		CZM model checker				UPPAAL	
Payload	Correct	Steps	Zones	Time	Memory	Time	Memory
1	Yes	6566	1858	86 s	252 MB	23 s	88 MB
2	Yes	8606	2498	2 min	251 MB	69 s	205 MB
3	Yes	10423	3142	7 min	527 MB	2 min	325 MB
4	Yes	12143	3782	2 min	251 MB	3 min	436 MB
5	Yes	13863	4422	4 min	312 MB	4 min	563 MB
20	Yes	45029	14038	5 min	261 MB	18 min	2 GB
21	Yes	46750	14678	6 min	259 MB	18 min	2 GB
22	Yes	48470	15318	5 min	262 MB	19 min	2 GB
23	Yes	50190	15958	4 min	259 MB	20 min	3 GB
24	Yes	51991	16024	7 min	264 MB	22 min	3 GB
25	Yes	52629	16024	5 min	314 MB	22 min	3 GB
31	Yes	54309	16024	16 min	415 MB	29 min	4 GB
32	Yes	54589	16024	6 min	313 MB	31 min	4 GB
33	Yes	54869	16024	28 min	955 MB	31 min	4 GB
34	Yes	55149	16024	13 min	313 MB	MEMOUT	
60	Yes	66230	16024	18 min	520 MB	MEMOUT	
100	Yes	90230	16024	57 min	941 MB	MEMOUT	
150	Yes	120230	16024	30 min	406 MB	MEMOUT	
200	Yes	150230	16024	72 min	938 MB	MEMOUT	
262	Yes	187430	16024	28 min	413 MB	MEMOUT	

Table 2. Comparison of our prototype with UPPAAL and RED on the (timing-intensive) Fischer protocol benchmark.

	CZM model checker				UPPAAL		RED		
Benchmark	Steps	Zones	Time	Memory	Time	Memory	Steps	Time	Mem
Fischer 5	3156	1496	1 s	108 MB	0 s	37 MB	5	0 s	21 MB
Fischer 6	42528	17426	32 s	156 MB	0 s	37 MB	5	1 s	45 MB
Fischer 7	612531	227522	17 min	302 MB	1 s	37 MB	5	1 s	66 MB
Fischer 8	TIMEOUT				3 s	38 MB	5	3 s	105 MB
Fischer 9	TIMEOUT				15 s	42 MB	5	8 s	174 MB

This is however not surprising as the Fischer protocol does not fall into the class of systems whose verification our approach aims at. We presented a specialized technique for timed systems with a large discrete state space but only a few clocks, an important class of models that comprise, e.g., data-intensive asynchronous communication protocols. The Fischer protocol model, on the other

hand, has a large number of clocks (one per component), but only few locations, thus the standard semi-symbolic state space representation used in Uppaal is already quite effective here. Also, RED's symmetry reduction is beneficial for this particular protocol.

6 Conclusion and Outlook

DBMs and BDDs impressively demonstrate their effectiveness in model checkers such as Uppaal and NuSMV. However, since NuSMV can only handle pure discrete models and Uppaal does not have a symbolic representation for the discrete part of the state space, both tools fail in verifying timed systems with large discrete control structures.

This paper presented a fully symbolic approach to timed model checking that combines DBMs with BDDs. In contrast to other approaches, our technique neither suffers from a loss of modeling precision (we remain in the classical timed automata framework) nor leads to blow-ups in the BDDs (we avoid the encoding of timing interdependencies in the BDDs).

Inspired by the encouraging experimental results, in future work, we plan to extend the scope of our approach to arbitrary timed systems. A promising direction is to investigate efficient representations of sets of pairs of DBMs and BDDs. So far, our prototype uses a simple hash map for assigning *complete* DBMs to BDDs. However, for many problem instances, considering *partial* DBMs might be more appropriate as it gives more flexibility in finding efficient representations.

Acknowledgment. This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
4. Behrmann, G., Larsen, K.G., Pearson, J., Weise, C., Yi, W.: Efficient timed reachability analysis using clock difference diagrams. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 341–353. Springer, Heidelberg (1999)
5. Bengtsson, J.: Clocks, DBM, and States in Timed Systems. PhD thesis, Uppsala University (2002)
6. Beyer, D.: Improvements in BDD-based reachability analysis of timed automata. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 318–343. Springer, Heidelberg (2001)
7. Beyer, S., Böhm, P., Gerke, M., Hillebrand, M.A., der Rieden, T.I., Knapp, S., Leinenbach, D., Paul, W.J.: Towards the formal verification of lower system layers in automotive systems. In: ICCD, pp. 317–326. IEEE Computer Society, Los Alamitos (2005)

8. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from psl. Electr. Notes Theor. Comput. Sci. 190(4), 3–16 (2007)
9. Bozga, M., Maler, O., Pnueli, A., Yovine, S.: Some progress in the symbolic verification of timed automata. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 179–190. Springer, Heidelberg (1997)
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
11. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2), 142–170 (1992)
12. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model checker. STTT 2(4), 410–425 (2000)
13. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
14. Dill, D.L., Wong-Toi, H.: Verification of real-time systems by successive over and under approximation. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 409–422. Springer, Heidelberg (1995)
15. FlexRay Consortium: FlexRay Communications System Protocol Specification Version 2.1 Revision A (2005)
16. Møller, J.B., Lichtenberg, J., Andersen, H.R., Hulgaard, H.: Fully symbolic model checking of timed systems using difference decision diagrams. Electr. Notes Theor. Comput. Sci. 23(2) (1999)
17. Pigorsch, F., Scholl, C., Disch, S.: Advanced unbounded model checking based on AIGs, BDD sweeping, and quantifier scheduling. In: FMCAD, pp. 89–96. IEEE Computer Society, Los Alamitos (2006)
18. Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley (1992)
19. Seshia, S.A., Bryant, R.E.: Unbounded, fully symbolic model checking of timed automata using boolean methods. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 154–166. Springer, Heidelberg (2003)
20. Somenzi, F.: CUDD: CU Decision Diagram package release 2.4.2 (2009)
21. Wang, F.: Efficient verification of timed automata with BDD-like data structures. STTT 6(1), 77–97 (2004)
22. Yamane, S., Nakamura, K.: Development and evaluation of symbolic model checker based on approximation for real-time systems. Systems and Computers in Japan 35(10), 83–101 (2004)
23. Yovine, S.: Kronos: A verification tool for real-time systems. STTT 1(1-2), 123–133 (1997)

Comparison of Model Checking Tools for Information Systems

Marc Frappier, Benoît Fraikin, Romain Chossart,
Raphaël Chane-Yack-Fa, and Mohammed Ouenzar

GRIL, Université de Sherbrooke, Québec, Canada
benoit.fraikin@usherbrooke.ca

Abstract. This paper compares six model checkers (ALLOY, CADP, FDR2, NUSMV, PROB, SPIN) for the validation of information system specifications. The same case study (a library system) is specified using each model checker. Fifteen properties of various types are checked using temporal logics (CTL and LTL), first-order logic and failure-divergence (FDR2). Three characteristics are evaluated: ease of specifying information system i) behavior, ii) properties, and iii) the number of IS entity instances that can be checked. The paper then identifies the most suitable features required to validate information systems using a model checker.

1 Introduction

Information systems (IS) now play a prominent role in our society to support business processes and share organisational data. Yet, even if they are one of the early application domains of computing, their development relies mostly upon a manual and informal process. The problem addressed in this paper is the formal *validation* of IS specifications using model checking. Model checking is an interesting technique for IS specification validation for several reasons: it provides broader coverage than simulation or testing, it requires less human interaction than theorem proving, and it has the ability to easily deal with both safety and liveness properties.

The validation of IS specification is of particular interest in model-driven engineering (MDE) and generative programming, which aim at synthesizing an implementation of a system from models (i.e., specifications). Hence, if the synthesis algorithms are correct, one only needs to validate the models to produce correct systems. IS MDE specification languages usually do not have any dedicated model checker. Since developing a model checker is a long process and since several model checkers already exist, it is simpler to choose an existing tool that is maintained by a team specialized in the model checking field. In this paper, we compare six model checkers: SPIN [11], NUSMV [4], FDR2 [18], CADP [10], ALLOY [12] and PROB [13], which are representative of the main classes of model checkers: explicit state, symbolic, bounded and constraint satisfaction. The comparison is based on a single case study which is representative of IS structure and properties. Our comparison aims at answering the following questions.

1. Is the modeling language of the model checker adapted for the specification of IS models? This is especially important in the context of MDE IS, since it must be

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 581–596, 2010.
© Springer-Verlag Berlin Heidelberg 2010

straightforward to automatically translate an IS MDE specification into the language of the model checker.
2. Is the property specification language adapted to specify IS properties?
3. Is the model checker capable of checking a sufficient number of instances of IS entities?

Our case study focuses on the control part of an IS, which determines the sequences of actions that the IS must accept. Validation of input-output behavior (data queries) and user interactions with graphical user interface are omitted.

This paper is structured as follows. Section 2 presents a synthesis of related work on model checking of IS. Section 3 presents a description of the case study, a library IS. Section 4 provides an overview of the model checkers, comparing relevant points. The modeling and verification process of the IS for each tool is provided in Sect. 5. Then, the analysis of processes and the model checking results for the case study are presented in Sect. 6. Finally, we conclude in Sect. 5.

2 Related Work

There is an extensive literature on model checking. This section focuses on comparative studies of model checkers related to IS. Model checking has been extensively applied to business process modelling. Yeung [20] proposes a framework to analyse *suspendible business* transactions modelling with statecharts and CSP [18]. It is applied to a library specification similar to the one studied in this paper. However, the paper does not actually experiment with model checkers to check business process. In [2], a travel agency business process has been modelled with SPIN and PROMELA, and CIA and CSP (LP). Safety properties and deadlocks have been successfully verified with both model checkers; reachability properties have not been tested. The authors propose an extension of CSP with notion of "compensation" (a behavior to compensate a process failure). In [16], business processes are converted from BPEL to automata, but also to Petri nets and CSP and LOTOS [5]. It concludes that process algebras are suitable for verification of the reliability of IS, in the particular case of business process. In [8], the authors study the verification of data-driven applications in the particular case of web-based systems using an ASM-like [19] specification language. The study focuses particularly on reachability properties, but any type of property can be used for modelling. The modelling process is complex and demands significant expertise. Both modelling techniques give an insight on what can be done with these subclasses of IS. In [3], four state-based model modeling techniques with their model checkers (USE, Alloy, ZLive and ProZ) are compared along four criteria: animation, generation of pre and postconditions, execution analysis and expertise. The study mostly checks invariant properties.

Each of these studies provides partial answers to our questions, for a subset of model checkers, using different case studies and properties. This makes it difficult to compare model checkers and identify the one best suited for IS validation.

3 Presentation of the Case Study

This section presents the user requirements of a library system which is used for the formal verification of properties. In order to avoid any confusion, key concepts are first defined. *Lending* a book means that a user borrows a book without reserving it beforehand. *Taking* a book means borrowing a book after having reserved it. *Borrowing* a book denotes either *taking* or *lending* it. In the requirements list, a *member* is a person who has *joined* (and still not *left*) library membership.

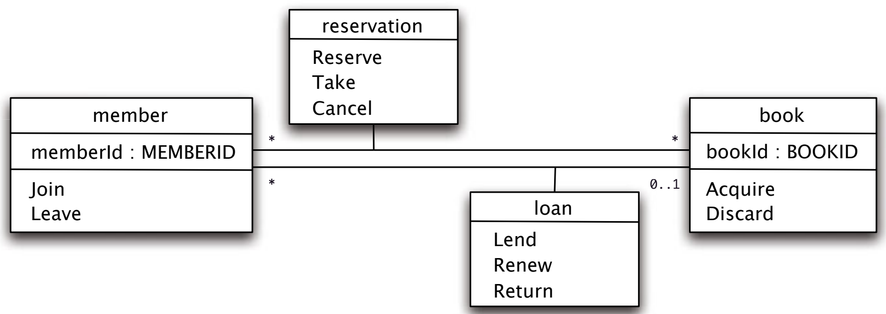


Fig. 1. Requirement class diagram of the library system

The requirement class diagram corresponding to the model is given in Fig. 1. Entity attributes are listed in the upper part of each class, while entity actions are listed in the lower part. The library system only contains two entity types: **books** and **members**. **Members** can `Join` and then `Leave` library membership whereas **books** can be `Acquired` and then `Discarded`. **Members** can `Lend`, `Renew` several times and `Return` a **book**. They can also `Reserve` a **book** under certain conditions (e.g. if it cannot be lent at that moment), and then, either `Cancel` the reservation or `Take` the **book**. Hence the library system contains 10 actions.

The following list describes the properties that we verify using the model checkers.

1. A book can always be acquired by the library when it is not currently acquired.
2. A book cannot be acquired by the library if it is already acquired.
3. An acquired book can be discarded only if it is neither borrowed nor reserved.
4. A person must be a member of the library in order to borrow a book.
5. A book can be reserved only if it has been borrowed or already reserved by another member.
6. A book cannot be reserved by the member who is borrowing it.
7. A book cannot be reserved by a member who is reserving it.
8. A book cannot be lent to a member if it is reserved.
9. A member cannot renew a loan if the book is reserved.
10. A member is allowed to take a reserved book only if he owns the oldest reservation.
11. A book can be taken only if it is not borrowed.
12. A member who has reserved a book can cancel the reservation at anytime before he takes it.

13. A member can relinquish library membership only when all his loans have been returned and all his reservations have either been used or canceled.
14. Ultimately, there is always a procedure that enables a member to leave the library.
15. A member cannot borrow more than the loan limit defined at the system level for all users.

In the context of IS, one usually distinguishes two types of properties. The first one is called a *liveness* property. It represents the fact that the system is still alive, i.e. not stuck in a deadlock (the system is blocked in a single state) or a livelock (the system loops in a subset of states considered as non-evolving). They can also express the fact that an action implies a reaction from the system; the latter is however rarely used in information systems, where actions are human-driven (one cannot force a user to trigger specific actions). Properties 1, 12 and 14 are liveness properties. In IS, liveness properties usually describe *sufficient conditions* to *enable* an action (immediately or sometime in the future). For instance, Property 1 states that the library *has the right* to acquire the book (under certain conditions). In other terms, it forces the IS to allow the action, but the user is not forced to invoke this action. The other properties are called *safety* properties. They usually describe *necessary conditions* to enable an action, or what a user is *not allowed* to do with the system at a given point. The remaining properties are safety properties. A third type of properties is usually distinguished from these two. These are *fairness* properties, but as IS users cannot be forced to do some action, they seldom occur in IS specifications. Fairness is not considered in this study.

4 An Overview of the Model Checkers

Four large families of model checkers are considered. *Explicit state* model checkers, like CADP, SPIN and FDR2, use an explicit representation of the transition system associated to a model specification. This transition system is either computed prior to property verification, as in CADP and FDR2, or on-the-fly while checking a property, as in SPIN (also possible in some cases with CADP and FDR2). *Symbolic* model checkers, like NUSMV, represent the transition system as a Boolean formula. *Bounded* model checkers, like NUSMV and ALLOY (indirectly), consider traces, of a maximal length k, of the transition system and represent them using a Boolean formula. *Constraint satisfaction* model checkers, like PROB, use logic programming to verify formula. SPIN, CADP, NUSMV and PROB support temporal languages (LTL [17], CTL [7] and XTL [14]) for property specification. ALLOY and FDR2 use the same language for both model specification and property specification (first-order logic and CSP, respectively).

4.1 SPIN

SPIN was one the first model checker developed, starting in 1980. It introduced the classical approach for on-the-fly LTL model checking. Specifications are written in Promela and properties in LTL. An LTL property is compiled in a Promela `never` claim, i.e. a Büchi automaton. SPIN generates the C source code of an on-the-fly verifier. Once compiled, this verifier checks if the property is satisfied by the model.

Working on-the-fly means that SPIN avoids the construction of a global state transition graph. However, it implies that transitions are (re-)computed for each property to verify. Hence, if there are *n* properties to verify, a transition is potentially computed *n* times, depending on optimizations.

PROMELA, the model specification language of SPIN, is inspired from C. Hence, it is an imperative language, with constructs to handle concurrent processes. State variables can be global and accessed by any process. PROMELA offers basic types like `char`, `bit`, `int` and arrays of these types. Processes can communicate by writing and reading over a *channel*, either synchronously using a channel of length 0, or asynchronously, using a channel of length greater than 0. Operator `atomic` allows a compound statement to be considered as a single atomic transition, except when this compound contains a blocking statement, such as a guard or a blocking write or read over a channel, in which case the execution of the `atomic` construct can be interrupted and control transferred to another process. Statements can be labeled and these labels can be used in LTL formulae.

SPIN uses propositional LTL, with its traditional operators *always*, *eventually* and *until*. The latter is sometimes referred as the "strong until" operator, as opposed to the "weak until" operator. The *next* operator is not allowed to ensure that partial order reduction can be used during the model checking. An LTL formula can refer to labels and state variable values of a PROMELA specification. SPIN only considers states; there is no notion of an event on a transition. An LTL formula holds for a PROMELA specification if and only if it holds for every possible run of the PROMELA model. A run is an execution trace consisting of the sequence of states visited during execution. It can be infinite.

4.2 NUSMV

NUSMV is a model checker based on the SMV (Symbolic Model Verifier) software, which was the first implementation of the methodology called *Symbolic Model Checking* described in [15]. This class of model checkers verifies temporal logic properties in finite state systems with "implicit" techniques. NUSMV uses a symbolic representation of the specification in order to check a model against a property. Originally, SMV was a tool for checking CTL properties on a symbolic model. But NUSMV is also able to deal with LTL (+Past) formulae and SAT-based *Bounded Model Checking*. The model checker allows to write properties specification both in CTL or LTL and to choose between BDD-based symbolic model checking and bounded model checking.

NUSMV uses the SMV decription language to specify finite state machines. A specification consists of module declarations and each module may include variable declarations and constraints. System transitions are modelled by assignment contraints or transition constraints, which define next values for declared variables in a module. An assignment gives explicitly a value for a variable in the next step, while a transition constraint, given by a boolean formula, restricts the set of potential next values. Each module can be instantiated by another one, for example by the main module, as a local variable. In fact, each instance of a module is by default processed synchronously with the others during an execution. But NUSMV can also model interleaving concurrency by using the "process" keyword in module instantiation. To get different instances

of a module, instantiations can be parameterized. However, the description language is quite low-level. All assignments, parameters or array indexes have to be constant. Thus, specifications may be longer than in PROMELA, because each case has to be explicitly written. As NUSMV modules can declare state variables and input variables, an SMV specification can be both state or event oriented. Input variables are used to label incoming transitions and their values can only be determined by specifying transition constraints.

CTL properties can only refer to state variables; LTL properties can refer to both state variables and input variables. Moreover, NUSMV can also check invariant properties, which can be written in a temporal manner as *Always p* where *p* is a boolean formula. Invariant specifications are checked by a specialized algorithm during reachability analysis, that gives a result faster than CTL or LTL algorithms.

4.3 FDR2

FDR2 is an explicit state model checker for CSP, the well known process algebra. FDR2 can check refinement, deadlocks, livelocks and determinacy of process expressions. It gradually builds the state-transition graph, compressing it using state-space reduction techniques, while checking properties, which also makes it an *implicit* state model checker.

Models are described using a variant of CSP, called CSP_M. It supports classical process algebra operators like prefix, choice, parallel composition with synchronisation, sequential composition and guards. Quantified versions of choice, parallel composition and sequential are supported. FDR2 supports basic data types like integer, boolean, tuples, sets and sequences. Lambda terms can be used to define functions on these types. Expressions are dynamically typed (except for actions, called channels in CSP, which are declared and typed). CSP does not support state variables; however, they can be simulated to some extent by using a recursive process with parameters.

Properties are expressed as CSP processes. They are checked using process refinement. FDR2 supports three refinement relations: $\sqsubseteq_T$ (trace refinement), $\sqsubseteq_F$ (stable-failure refinement) and $\sqsubseteq_{FD}$ (stable-failure-divergence refinement). We say that $P \sqsubseteq_T Q$ iff the traces of Q are included in the traces of P. A trace of a process P is a sequence of visible events that P can execute. We say that $P \sqsubseteq_F Q$ iff the failures of Q are included in the failures of Q. A failure (t, S) of a process P denotes the set of events S that P can refuse after executing trace t. Trace refinement is used to check safety properties, while stable failure is used to check liveness (or reachability) properties. Failure-divergence refinement is used to check livelocks (infinite loops on internal actions), which are not relevant for our case study.

4.4 CADP

CADP is a rich and modular toolbox. We have selected LOTOS-NT to specify models and XTL to specify properties. The XTL model checker takes as input a labelled transition system (LTS), encoded in the BCG (*Binary Coded Graph*) format. LOTOS-NT is a variant of LOTOS that supports local states variables. A LOTOS-NT specification is translated into into a LTS using Caesar. This LTS is minimized into a trace equivalent

LTS. Finally, properties written in XTL are checked against this LTS using the XTL model checker.

LOTOS-NT is inspired from LOTOS. A LOTOS-NT specification is divided into two complementary parts: an algebraic specification of the abstract data types and a process expression. LOTOS-NT offers traditional process algebra operators like sequence, choice, loops, guard and parallel synchronization. It supports state variables, which are local to a process and cannot be referred by another process. Assignment statements can be freely mixed with other process expression constructs.

XTL, the property specification language of CADP, is used to express temporal logic properties. XTL provides low-level operators which can be used to implement various temporal logics like HML, CTL, ACTL, LTAC, as well as the modal mu-calculus. XTL formulae are evaluated on a LTS. XTL allows one to refer to transitions (events) and values of their parameters. No LTL library is currently provided. In this paper, the CTL and HML libraries are used.

Since the LTS does not contain any state variable, the difficult part in writing XTL properties for LOTOS-NT models is to characterize states. Indeed, the specifier can only use action labels to define particular states. The HML library, with its two handy operators *Dia* and *Box*, is used for this purpose. $Box(a, p)$ holds in a given state if and only if every action matching action pattern a leads to a state matching state pattern p. On the other hand, $Dia(a, p)$ holds for a given state if and only if there exists at least one action matching action pattern a leading to a state matching state pattern p. An XTL formula holds for a LTS if and only if it holds for all states of the LTS.

4.5 ALLOY

ALLOY is a symbolic model checker. Its modeling language is first-order logic with relations as the only type of terms. Basic sets and relations are defined using “signatures”, a construct similar to classes in object-oriented programming languages, which supports inheritance. ALLOY uses SAT-solvers to verify the satisfiability of axioms defined in a model and to find counterexamples for properties (theorems) which should follow from these axioms.

An ALLOY specification consists of a set of signatures, noted (`sig`), which basically define sets and relations. Constraints, noted `fact`, are formulae which condition the values of sets and relations. The declaration `sig X {r : X -> Y}` declares a set `X` and a ternary relation `r` which is a subset of the Cartesian product $X \times X \times Y$. ALLOY supports usual operations on relations, like union, intersection, difference, join, transitive closure, domain and range restriction. Integer is the only predefined type. Cardinality constraints can be defined on relations (*e.g.*, injections and bijections). Properties are simply written as first-order formulae.

4.6 PROB

PROB is a model checking and an animation tool designed for the B Method [1]. Currently it also supports CSP_M, Z, and Event-B. This study uses the B Method and the B language.

B specifications are organized into abstract machines (similar to classes and modules). Each machine encapsulates state variables, an invariant constraining the state variables, and operations on the state variables. The invariant is a predicate in a simplified version of the ZF-set theory, enriched by many relational operators. In an abstract machine, it is possible to declare abstract sets by giving their name without further details. This allows the actual definition of types to be deferred to implementation. Operations are specified in the Generalized Substitution Language, which is a generalization of Dijkstra's guarded command notation. Hence, operations are defined using substitutions, which are like assignment statements. A substitution provides the means for identifying which variables are modified by the operation, while avoiding mentioning those that are not. The generalization allows the definition of non-deterministic and preconditioning substitutions. The preconditioning substitution is of the form **PRE** *P* **THEN** *S* **END**, where *P* is a predicate and *S* a substitution. When *P* holds, the substitution is executed; otherwise, the result is undetermined and the substitution may abort.

Properties in PROB can be written in LTL, past LTL or CTL, hence combining the strengths of each language. In addition, PROB allows for the inclusion of first-order formulae in temporal formulae. It also offers two convenient operators for LTL. The first one, denoted by $e(A)$, checks if the action A is executable in a given state of a sequence. The second one, denoted by $[A]$ checks if A is the next operation in the sequence. Consequently PROB can express properties on either states or events.

5 Specifying the Model and the Properties

This section describes how the IS model and properties are specified with each model checker. For sake of conciseness, specifications are omitted. They are available in [6].

5.1 SPIN

Two styles have been considered for the SPIN specification. In the first style, there is only one process which loops over a choice between all actions. It was quickly abandoned, because it blows up quite rapidly in terms of number of states. In the second style, there is one process for each instance of each entity and each association. The process describes the entity (or the association) life cycle. Therefore, the PROMELA specification of the case study contains four process definitions, one for each entity (book and member) and one for each association (loan and reservation). Each process definition is instantiated n_i times to model n_i instances of entity i, and $n_i * n_j$ times to model an association between entity i and j.

We use a producer-modifier-consumer pattern as the basis of a life cycle for an entity and an association. It can be represented by the following regular expression $P.M^*.C$ where P is the producer (for example `Acquire`), M is a modifier and C is a consumer (for example `Discard`). The concatenation operator "." of regular expressions can be represented by a semi-colon "`;`", the sequential composition operator of PROMELA, or an arrow "`->`" that denotes the same operator. Some events have a precondition which is not represented in the regular expression. For example, a book cannot be discarded if it is still borrowed. Consequently the execution of an event is guarded by a precondition.

When a member takes a book he has reserved, two associations are involved: the loan association and the reservation association. This leads to ensure that both processes execute the `take` event in an atomic step. It is not obvious and straightforward. To achieve an atomic step, the `take` event is split into two events: one in the reservation association process (as a consumer) and one in the loan reservation (as a producer). A channel with an empty capacity is used to ensure the handshake. This is a classic strategy in PROMELA. Nevertheless, the handshake cannot be made within an `atomic` instruction. This could break the local atomicity in the sender. But it could be used at the end of an `atomic` and at the beginning of another. In this way, no other process can be interleaved with the handshake of the two processes. The result is a pattern described in [6], in which an event is simultaneously the consumer of an association and the producer of another one.

In SPIN, the properties are expressed using LTL. Reachability properties are difficult to express in LTL. Fortunately, since event preconditions are explicitly written via labels in the specification, expressing a property like "a book can be acquired" is straightforward. Consequently, when a property asserts the possibility of an event execution, it is represented by a propositional formula in the LTL formula that uses a label of the process and, sometimes, the precondition of this event. For example, "the book `b0` can be acquired" is expressed as "process book `b0` is at the `discarded` label".

Property 14 is not expressible in LTL, since it is equivalent to a reset property. The reset property is known to be expressible in CTL only. LTL and CTL are complementary languages. The semantics of LTL formulae is defined on traces of the transition system, while the semantics of CTL formulae is defined on the transition system itself, which allows one to refer to the branching structure of the execution. Some properties can only be expressed in either LTL or CTL. For instance, a *reset* property, which states that it is always possible to go back to some desired state, cannot be expressed in LTL, since this transition to reset does not have to occur in each possible run of the system. Since an LTL formula holds if and only if it holds for every possible run of the system, an LTL property would force this reset transition to occur in every run. Dually, a property of the form "eventually p always holds" cannot be expressed in CTL [9], due to the branching nature of the logic.

Two simple patterns can express almost 70 % of the requirements. In LTL and with the state-oriented paradigm, these patterns are expressed as "if action A can be executed then the state verifies P" or "if P holds then action A can be executed" where P is only true between actions B and C. Therefore the two main patterns are $\square\,(\texttt{can_}A \Rightarrow P)$ or $\square\,(P \Rightarrow \texttt{can_}A)$. Inexpressible properties cannot simply be considered as negligible. This is an important weakness of SPIN that cannot be overcome.

5.2 NuSMV

To model the library system example in an SMV specification, we use a systematic method based on the structure of the class diagram. Each class, that represents an object and has attributes, is encoded into a module containing variables and parameterized by a key to identify entities. Then, for each kind of action defined in the system, a new module is created, parameterized by class modules involved in that action. Action

modules check that a given precondition is satisfied. Then, if the precondition holds, they modify variables of entities to apply postconditions using assignment constraints.

Properties of the library system can be expressed by CTL formulae on state variables, or by using LTL formulae with state variables or input variables. Specifications on state variables are close to Promela specifications, except that NuSMV can check CTL and LTL properties. This allows to easily express all requirements. Specifications on input variables are event-oriented. However, only LTL can be used to write event-oriented specifications in NuSMV.

Property 1 is a sufficient condition to enable an event. It is easily expressed as follows: `AG (!book1.is_acquired -> EX book1.is_acquired)`. Property 12 is specified in a similar manner, except that it must be repeated for each position in the array representing the reservation queue. Hence, the text of properties may linearly grow with the number of entity instances, an unfortunate limitation due to the restriction to constants in accessing array positions. Property 14 is also very similar to 1, except that `EF` is used instead of `EX`. Properties expressing necessary conditions (2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13) can be expressed in two different forms (state, event). For instance, property 2 is expressed using events, saying that a discard must always occur between two acquires:

```
G acquire1.do ->
      X((!acquire1.do U discard1.do) | G !acquire1.do)
```

5.3 FDR2

CSP is quite handy to explicitly represent IS entities life cycles. We use a nominal/controller pattern to specify the library case study. Each entity E is represented by a quantified interleave of the form $E = |||k : T @ E(k)$, where $E(k)$ models the life cycle of an instance k of entity E. The associations to which an entity E participates are represented in a similar manner, and called within $E(k)$. The global behavior is obtained by composing the entities in parallel: $Nominal = E_1 \parallel \ldots \parallel E_n$. Since CSP$_M$ does not directly implement Hoare's parallel operator $\parallel$, we use CSP$_M$'s operator $\underset{X}{\parallel}$, where X denotes the association actions on which entities must synchronize. Some ordering constraints are represented using recursive processes to simulate state variables. The relevant state variables $\boldsymbol{v}$ of an entity E are represented by a recursive process $C_E(k, \boldsymbol{v})$ which offers a choice [ ] between actions to control, in the form $[\,]_i\ G \,\&\, a_i \rightarrow C_E(k, e(\boldsymbol{v}))$, where "&" denotes a guard operator with condition G which tests the values of $\boldsymbol{v}$. These control processes are composed in parallel with *Nominal*, synchronizing on a_i.

Safety properties are checked by trace refinement and are relatively easy to specify. Suppose that property p only involves actions a_i. One writes a process P that represents the traces on a_i satisfying p, and checks that the system Q, restricted to actions a_i, trace refines P as follows: $P \sqsubseteq_T Q \setminus (\Sigma - \{a_i\})$, where \ is the hiding operator of CSP$_M$ and Σ denotes all actions of the specification and "−" is set difference.

Reachability properties 1, 12 and 14 are checked using stable-failure refinement, and are a bit more tricky to specify. For instance, property 1 states that a book can always be acquired, if it is not currently in the library. This is typically specified in CSP as follows: $P \,|||\, CHAOS(\{b_i\}) \sqsubseteq_F Q \setminus \{c_i\}$. Process P recursively loops over acquire and discard events: $\mathsf{acquire}(b) \rightarrow \mathsf{discard}(b) \rightarrow P(b)$. Essentially, P states that discard can

never be refused after an acquire, and an acquire can never be refused after a discard. Hidden actions $\{c_i\}$ are those that unavoidably can occur between acquire and discard, *ie*, association actions. The interleave with $CHAOS(\{b_i\})$ states that other actions can occur before or after, but we do not really care about their order. Unfortunately, hiding association actions introduces unstable states, which weakens the specification of the property under stable-failure refinement. To make a short explanation, infinite internal action loops are introduced by hiding; hence some errors in the behavior of association actions are not detected by this form of property specification. To overcome this, we have to check each association in isolation, disabling events from the other associations, which is weaker than property 1. These are very subtle issues which are difficult to master. Reachability properties of IS specifications are far from trivial to specify in CSP.

5.4 CADP

The LOTOS-NT specification of the library system is similar in structure to the CSP specification already described. Since there is no quantified interleave operator in LOTOS-NT, one has to hardcode entities and associations interleaves, which means that the number of interleaves to hard code in the specification text grows exponentially with the number of entities, making verification experiments a bit cumbersome.

Safety properties of the case study are defined using two patterns. The first states that an action `A` should not happen between two actions `B` and `C`. For example, a member should not leave the library if he has reserved a book (i.e. between a `Reserve` and a `Take` or `Cancel`). The second pattern expresses the prohibition of an action `A` outside a sequence delimited by two actions `B` and `C`; it is illustrated by the fact that a member should not renew a book if he has not borrowed it yet (i.e. outside a `Lend` or `Take` and a `Return`). In XTL, one can represent these patterns using macros. They are defined using a weak until operator, defined by macro `AW_A_B`. These two patterns, respectively called `no_A_between_B_and_C` for and `no_A_outside_B_and_C`, are used for properties (2,3,6,7,8,9,11,13) and 4, respectively. Liveness properties are written directly with classic ACTL and HML operators, like properties 1, 12 and 14. No correct formulation has been found for properties 5 and 10. For property 5, one must characterize using events the states where a book is *not borrowed nor reserved*. Property 10 involves a queue, which is as hard to describe using events.

5.5 ALLOY

Each IS entity E is represented by a signature `E`, which models the set of possible entity instances. System states are represented by a signature `sig S { e1 : E1, ..., en : En, a1 : Ei -> Ej, ...}`, where $\mathtt{e}_i$ models the active instances of E_i in a state, and $\mathtt{a}_i$ models the instances of association A_i. Each action is represented by a predicate $P[s : S, s' : S, p : T]$ relating a before-state s to an after-state s' for input parameters p. We have systematically followed a pattern for these predicates, which is a conjunction of a precondition, a postcondition and a "nochange" predicate that determines which attributes are unchanged by the action.

A property of the form "when condition C holds, action a must be executable" (*e.g.*, Property 1) is written as follows: $\forall s : S, p : T \cdot C \Rightarrow preA[s,p]$, where $preA[s,p]$ is the precondition of action $a(p)$. Similarly, if C is the result of executing an action $b(p)$ that should enable an action $a(p)$ (*e.g.*, Property 12), it can be written as $\forall s, s', p : T : S \cdot b[s,s',p] \Rightarrow preA[s',p]$. A property of the form "action a is executable only when condition C holds" (*e.g.*, Properties 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13) is written as: $\forall s : S \cdot preA[s,p] \Rightarrow C$. These three patterns are approximations of the property, because an action can be executed when its precondition holds and when $\exists s' : S \cdot postA[s,s',p]$ holds. In other words, the precondition must hold and the postcondition must be feasible. However, checking such an existential formula over IS states in ALLOY is usually not possible, unless the system model is small enough to fit within memory bounds; if the bounds provided for the check command do not include the full state space, then ALLOY can always find a small model where the desired state s' is not included. Luckily, the feasibility of postconditions is rarely an issue. Hence, we safely approximate the executability of an action to its precondition only. Invariant properties I for some action a (*e.g.*, Property 15) are easily expressed as a formula of the form $\forall s, s' : S, p : T \cdot I[s] \wedge a[s,s',p] \Rightarrow I[s']$. Properties expressing the reachability of a state from a condition C (*e.g.*, Property 14) are more tricky to express. We first tried to find a trace in the transition system, but that reveals to be impossible to check due to memory limitations. We then resorted to show the existence of a trace by describing how it can be computed. For property 14, the predicate describing such a trace states that an iteration over actions `return`, `cancel` and `leave` ultimately leads to a state where the member does not exist. If the property fails, it is either because the bound given for the length of the trace is too small (*i.e.*, the last state of the trace satisfies the precondition of a `return`, a `cancel` or a `leave`) or because the property is false in the model. By looking at the counterexample found, we can determine which case holds and increase accordingly the bounds for the trace and the number of library states. An additional difficulty is to determine the valid library states where C holds. These can be either characterized by a fact, which is error-prone to specify, but more efficient to check, or by executing entity and association producers from the initial state of the system to automatically construct valid library states satisfying C, which is simple to specify, but significantly less efficient to check.

5.6 PROB

Each action is specified as an operation defined as a precondition and a postcondition. Therefore, the main difficulty is to translate the ordering constraints (like, for example, "a book must be acquired in order to borrow it") in a precondition and find appropriate updates of state variables, as in SMV and ALLOY.

As already mentioned, most of the requirement can be categorized in two patterns: $\Box(\texttt{can_}A \Rightarrow P)$ and $\Box(P \Rightarrow \texttt{can_}A)$. In general, a requirement that looks like "A can be executed only if P is true" (the first template) can be seen as an indication that the precondition of A implies P. On the other hand, "A can be executed if P is true" (the second one) means that P implies the precondition of A. In PROB, `can_A` is expressed with the executability operator $e(A)$. However, it denotes the exact condition under which A is executable; it is not an approximation as we have done in ALLOY.

Specifying properties is straightforward using LTL and CTL. All properties are expressed in LTL, except 14, expressed in CTL. Property 12 is slightly more difficult to express. It denotes an ordering constraint that depends on both the current state (the book has been reserved by the member) and the previous action (once a `Take` is executed, the executability of the `Cancel` is not needed anymore). Thus, the executability operator, the *next action* operator and the LTL release operator are needed. This property does not not fit in the two described patterns.

Since PROB uses the B notation, it can be used in conjunction with Atelier B. This means that some proofs can be done prior or after using PROB. These tools can work together. For example, Property 15 is defined as an invariant. Atelier B generates proof obligations for invariants. But when the proof fails, PROB is quite useful to find where the problem is located. On the other hand, most temporal properties are generally not provable in Atelier B because they cannot be easily expressed as an invariant.

6 Analysis of the Case Study

In this section, we analyse the results of our case study along several aspects of IS specications which distinguish the salient features of each model checker.

Model specification language: abstraction over entity instances. This feature enables the specifier to parameterize the number of instances for each entity and association (*e.g.*, the number of books). If it is lacking, then the size of the specification text grows exponentially. All model checkers, except NUSMV and LOTOS-NT support this feature. It is worse in NUSMV, where each transition must be hardcoded for a given member and book. In LOTOS-NT, quantification for interleave is missing.

Model specification language: representation of entity and association structures. This is reasonably well supported by all model checkers. Modeling actions that involve several associations, like `take`, is not trivial in SPIN.

Model specification language: representation of IS scenarios. This is also reasonably well supported by all model checkers. IS requirements are often described as scenarios on events, from which event ordering constraints are deduced. These ordering constraints are more explicitly represented in event-based languages like CADP and SPIN. They are encoded as preconditions in state-based languages like SPIN, NUSMV, PROB and ALLOY, which are a little bit more cryptic.

Property specification language: abstraction over entity instances. Similarly, this feature enables the specifier to abstract from entity instances by using quantification on variables. If it is lacking, either the number of properties grows exponentially with the number of instances to check, or, as we did in this case study, a property is hardcoded for a particular instance of each entity, assuming that each entity behaves in a similar fashion (which may not hold in practice). NUSMV, LOTOS-NT and SPIN lack this feature, since it is generally not supported in LTL, CTL and XTL. PROB does not suffer from this limitation in CTL and LTL, because it evaluates properties for all elements of abstract sets when necessary. Hence, only PROB, FDR2 and ALLOY fully support this feature.

IS property specification. We have identified the following classes of properties for IS:

1. **SCE: Sufficient state condition to enable an event** (*e.g.*, case study properties 1 and 12). These are relatively easy to specify in state-based languages like NuSMV, ProB and Spin. All of these properties must be approximated in Alloy, otherwise they require a too large number of atoms to be completely checked. The validity of the approximation relies on the hypothesis that the postcondition of an action is satisfiable when the precondition holds. FDR2 can also handle these properties using stable failure refinement, but sometimes by approximation (property 1).
2. **NCE: Necessary state condition to enable an event** (*e.g.*, case study properties 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13). These are also relatively easy to specify in state-based languages like NuSMV, ProB and Spin, and with some approximations for Alloy (similar to SCE). Properties 5 and 10 have not been specified in XTL for CADP, because the states where too difficult to characterize using events only. There were no problem to specify these using trace refinement in FDR2. However, we suspect that there may be cases where characterizing states using only events may be difficult.
3. **SCEF: Sufficient state condition to enable an event on some execution path** (*e.g.*, property 14.) This is easy to specify in NuSMV and ProB, thanks to CTL. It is also possible for CADP and FDR2, when the state condition is easy to characterize using events. It can be specified in Alloy by providing the path of events leading to the desired event, when such a path does not exceed the number of atoms available. It is not possible to specify these properties in Spin, since they are not supported by LTL.
4. **INV: Invariant state property** (*e.g.*, property 15). All model checkers can handle these without particular problems.

Property specification language: access to states and events. Since most of the properties use both states and events, model checkers that support both, like ProB and Alloy, are simpler to use, since they can represent properties more explicitly (or directly) than the others. CADP and FDR2 being event oriented, handling states is sometimes cumbersome. Spin offers limited supported for events and we have used it extensively, similarly in NuSMV, but to a lesser extent.

Execution time and number of entity instances. Figure 2 shows the execution for the number of instances (number of books and number of members). The average time per property is also provided, since not every model checker can handle all properties. Overall, CADP, NuSMV, ProB and FDR2 cannot check, within reasonable bounds of time and memory, more than 3 instances for each entity for at least one property, although for some properties they can check a few more instances. Spin can handle up to 5 entity instances. Alloy is the most efficient model checker for IS for large number of entity instances. FDR2 is the most efficient for 3 instances; it fails due to memory limitations for more than 3 instances per entity. Alloy can handle up to 98 instances for all properties except 14 in less than a minute, because it only needs to cover a small subset of the state space to check these properties. Property 14 is checked for 8 members and 8 books in a few minutes. With the library case study, 3 instances is a minimum to check reservation queues of length greater than 1. Note that the latest release of ProB fails for 3 properties, due to some defects which have been reported to authors. This is why we only include the results of 12 properties in Fig. 2.

Step	SPIN		NUSMV	FDR2	CADP	ALLOY		PROB
Nb of Books/Members	3/3	5/5	3/3	3/3	3/3	3/3	8/8	3/3
Check Time	772.52	8645.6	3844.5	77.08	970.19	221.08	288.59	1094.4
Number of properties	14	14	15	15	13	15	15	12
Average (per property)	55.18	617.54	256.3	5.14	74.63	14.74	19.24	91.19

Fig. 2. Model checking duration in seconds for the properties of the library specification

Tools support Simulators are available in each method, which is very handy to discover specification errors. The simulator in NUSMV is not straightforward to use, because it is sometimes difficult to select the transition to execute.

7 Conclusion

We have presented a comparison of six model checkers for the verification of IS. The comparison is based on a case study of a typical IS. The study reveals that a good IS model checker has to be very polyvalent. To conveniently specify IS models and properties, it should support both states and events. Process algebraic operators are desirable to easily expressed IS scenarios, while state variables are handy to streamline specification of properties. CTL seems sufficient to handle most common properties. LTL is useful, but insufficient (*e.g.*, SCEF properties). A pure first-order logic like ALLOY is sufficient, but less intuitive in the case of SCEF properties. Given these characteristics, PROB seems to be the most polyvalent model checker for IS.

Since these conclusions are drawn from a single example, they must be further validated with additional examples. However, the library case study is sufficiently complex to exhibit a good number of characteristics found in most IS. It only contains two entities and two associations; large IS typically have hundred of entities and associations, but it seems quite reasonable to suppose that the verification of a property can be restricted to the entities and attributes involved. Hence, the properties checked in this case study are representative of typical IS properties.

Additional case studies would certainly find other limitations of these model checkers. For instance, our case study only addresses the sequence of actions that an IS must accept. It does not cover output delivery (*e.g.*, queries) and user interface interactions.

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Augusto, J.C., Ferreira, C., Gravell, A.M., Leuschel, M., Ng, K.M.Y.: The benefits of rapid modelling for e-business system development. In: ER Workshops, pp. 17–28 (2003)
3. Aydal, E.G., Utting, M., Woodcock, J.: A comparison of state-based modelling tools for model validation. In: TOOLS-Europe 2008, Switzerland. LNBIP, vol. 11, pp. 278–296 (2008)

4. Biere, A., Clarke, E.M., Cimatti, A., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
5. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. In: van Eijk, P.H.J., Vissers, C.A., Diaz, M. (eds.) The Formal Description Technique LOTOS, pp. 23–73. Elsevier Science Publishers B.V., Amsterdam (1989)
6. Chane-Yack-Fa, R., Fraikin, B., Frappier, M., Chossard, R., Ouenzar, M.: Comparison of model checking tools for information systems. Tech. Rep. 29, Universit de Sherbrooke (2010), http://pages.usherbrooke.ca/gril/TR/TR-GRIL-1006-29.pdf
7. Clarke, E.M., Emerson, E.A.: Synthesis of synchronization skeletons for branching time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131. Springer, Heidelberg (1981)
8. Deutsch, A., Sui, L., Vianu, V.: Specification and verification of data-driven web applications. Journal of Computer and System Sciences 73(3), 442–474 (2007)
9. Emerson, E.A., Halpern, J.Y.: “Sometimes” and “Not Never” revisited: On branching versus linear time temporal logic. J. ACM 33(1), 151–178 (1986)
10. Garavel, H.: Compilation et vrification de programmes LOTOS. Ph.D. thesis, Universit Joseph Fourier, Grenoble (November 1989)
11. Holzmann, G.J.: The Spin Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2004)
12. Jackson, D.: Software Abstractions. MIT Press, Cambridge (2006)
13. 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)
14. Mateescu, R., Garavel, H.: XTL: A meta-language and tool for temporal logic model-checking. In: Proceedings of the International Workshop on Software Tools for Technology Transfer STTT 1998, Aalborg, Denmark, p. 10 (July 1998)
15. McMillan, K.L.: Symbolic Model Checking. Ph.D. thesis, Carnegie Mellon University (1993)
16. Morimoto, S.: A survey of formal verification for business process modeling. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part II. LNCS, vol. 5102, pp. 514–524. Springer, Heidelberg (2008)
17. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57 (1977)
18. Roscoe, B.A.W.: The Theory and Practice of Concurrency, 3rd edn. Prentice Hall PTR, Englewood Cliffs (1998) (amended 2005)
19. Spielmann, M.: Abstract state machines: Verification problems and complexity. Ph.D. thesis, Bibliothek der RWTH Aachen (2000)
20. Yeung, W.L., Leung, K.R.P.H., Wang, J., Dong, W.: Modelling and model checking suspendible business processes via statechart diagrams and CSP. Science of Computer Programming 65(1), 14–29 (2007)

A Modular Scheme for Deadlock Prevention in an Object-Oriented Programming Model

Scott West, Sebastian Nanz, and Bertrand Meyer

ETH Zurich
firstname.lastname@inf.ethz.ch

Abstract. Despite the advancements of concurrency theory in the past decades, practical concurrent programming has remained a challenging activity. Fundamental problems such as data races and deadlocks still persist for programmers since available detection and prevention tools are unsound or have otherwise not been well adopted. In an alternative approach, programming models that exclude certain classes of errors by design can address concurrency problems at a language level. In this paper we review SCOOP, an existing race-free programming model for concurrent object-oriented programming, and extend it with a scheme for deadlock prevention based on locking orders. The scheme facilitates modular reasoning about deadlocks by associating annotations with the interfaces of routines. We prove deadlock-freedom of well-formed programs using a rigorous formalization of the locking semantics of the programming model. The scheme has been implemented and we demonstrate its usefulness by applying it to the example of a simple web server.

1 Introduction

Concurrent programming has remained a difficult task even for expert programmers, in spite of steady progress in the theory of concurrency. One possible reason is that concurrency is typically added to a language as a secondary concern, via thread libraries. These offer little support for a structured use of synchronization primitives, making it difficult for programmers to reason about their programs. Concurrency research has provided a set of tools, e.g. [13,18,6,3], for addressing the data races and deadlocks that arise from incorrect use of synchronization, but these do not tackle the source of the problem.

Another line of research therefore attempts to create languages that raise the level of abstraction for expressing concurrency and synchronization and hence to make programmers produce better code. Resulting programming models can also exclude certain classes of errors by construction, for example data races [8,1], usually accepting a penalty in performance or programming flexibility for the sake of program correctness.

One such programming model is SCOOP (Simple Concurrent Object-Oriented Programming) [15,16]. The model allows objects to be declared of a special type indicated by the keyword **separate**. Calls to routines (methods) on such separate objects are executed asynchronously, i.e. they will be spawned off to

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 597–612, 2010.
© Springer-Verlag Berlin Heidelberg 2010

```
class DEADLOCK
create set
feature
 x, y : separate S

 f
  do g (x) end

 g (a : separate S)
  do h (y) end

 h (b : separate S)
  do end

 set (a_x, a_y : separate S)
  do
   x := a_x
   y := a_y
  end
end
```

```
class MAIN
feature
 x, y : separate S

 run (d1, d2 : separate DEADLOCK)
  do d1.f; d2.f end

 make
  local
   d1, d2 : separate DEADLOCK
  do
   create x
   create y
   create d1.set (x, y)
   create d2.set (y, x)

   run (d1, d2)
  end
end
```

Program 1: SCOOP deadlock example

a thread separate from the current one. SCOOP then preserves freedom from object-level data races by requiring that separate objects be *controlled* before features can be invoked on them. Control is obtained by having such an object passed as an argument to a routine: within the scope of the routine, all separate objects that are its actual arguments are automatically locked. In Program 1, `x` is locked by the call to `g` in the body of `f`.

This locking behavior simplifies reasoning about concurrent programs for the programmer, as groups of concurrent objects are protected within the body of a routine and thus "sequential thinking" can be applied in this context. On the other hand, SCOOP offers no protection against deadlocks, a flaw shared with practically all concurrent programming languages.

In this paper, we extend SCOOP with a scheme for deadlock prevention, addressing a critical open problem of this programming model (indeed, Program 1 may deadlock). The scheme is based on establishing an order in which resources can be locked, hence preventing the formation of cyclical locking patterns. As the structure of locking in SCOOP is reflected in the call stack, annotations indicating the locking order are associated with the interfaces of routines, providing modularity at the routine level. We formalize the locking behavior of SCOOP using a structural operational semantics, providing the basis for the deadlock-freedom proof. We provide a technique for statically checking that programs are well-formed according to a well-formedness predicate, and prove that well-formed programs never deadlock. The technique has been implemented and applied to a simple web server programmed in SCOOP.

Other work in this area such as deadlock freedom for active objects in Java [11] provides less versatile structures (trees vs. orders). Techniques of similar power [3], however, are not grounded in an underlying language that is designed to make concurrent programming easier. Lastly, other partial operational semantics [17] only consider liveness properties in the light of model checking.

The remainder of this paper is structured as follows. In Section 2 we give an overview of SCOOP and reason on how deadlock can be detected both dynamically and statically. Section 3 provides a formalization of SCOOP's locking semantics. In Section 4 we describe our deadlock prevention scheme and prove that well-formed programs cannot deadlock. We describe related work in Section 5 and conclude in Section 6.

2 SCOOP Programs and Their Locking Semantics

SCOOP [15,16] is a programming model for concurrency, which can be implemented on top of any object-oriented language. Implementations are currently available for Eiffel [16,9] (the syntax we use in this paper) and Java [19]. In this section we first provide a short overview of the model, give a description of how deadlock may be identified, and finally show an annotation language for establishing a locking order among resources.

2.1 Overview of the Model

Asynchronous calls. The central idea of SCOOP is that every object is associated for its lifetime with a *processor*, an abstract notion denoting a site for computation: just as threads may be assigned to cores on a multi-core system, processors may be assigned to cores, or even to more remote processing units. The (unique) processor associated with a certain object is called its *handler*. Processors may handle multiple objects. A processor can be identified using its *processor tag*.

Processors are an abstraction, allowing the model to be mapped to multithreaded systems, distributed systems, or other concurrent architectures alike. For example in multithreaded systems every processor simply corresponds to one thread on the system, and the processor tag is the thread identifier. Whenever a new processor is created, a new thread is spawned.

Calls on an object are only executed by its handler. For example, if a processor p encounters a call `x.f`, and the object attached to `x` is handled by a processor q then p asks q to evaluate `x.f` on its behalf. If `x.f` does not return a result, processor p can continue executing concurrently with the computation taking place at q. If `x.f` returns a result, the runtime system makes sure that p waits for q to return the result before proceeding.

Type annotations. To make it clear for programmers which calls are executed asynchronously (invoked on objects residing on separate processors) and which calls are synchronous (invoked on objects residing on the current processor), the type system of SCOOP provides a special type indicated by the keyword **separate**: if a variable `x` is declared of separate type

```
x : separate X
```

then at creation of `x` with the statement

```
create x
```

a new processor p is created in addition to an object o of type X, and the handler of o is set to p. The type system also allows that the processor tags can be explicitly specified as in

```
x : separate <p> X
y : separate <p> Y
```

which at creation time would place objects x and y on the same processor p. These processor annotations have the scope of a class if applied to attributes of the class, and of the routine's body if applied to local variables of a routine.

Locking behavior. In order to prevent object-level data races in SCOOP, processors that are needed for the execution of a routine are automatically locked by the runtime system before entering the body of the routine; the locks are released upon the completion of the execution of the body. Thus all handlers of separate objects that occur in the body need be locked. The model prescribes that these separate objects need to be *controlled* (passed as arguments to the routine). At routine invocation the runtime system tries to lock the separate arguments' handlers: if the locking succeeds, the execution proceeds into the body of the routine; if it fails because one or more of the handlers are locked by other processors, the runtime system schedules the call to be retried later. In Program 1 the body of the feature f contains the command g(x), the locking behaviour described above would be seen here, as this call is invoked, requesting and locking the object x.

2.2 Deadlock in SCOOP

Knowing how locks and requests appear in the SCOOP model, we can now describe how a deadlock state may be detected. A deadlock state, based on waiting for resource availability as in [5], can be identified

- dynamically: construct a "waits-for" relation; if an element is related to itself in the transitive closure of such a relation, then the system is in a deadlock state. In the setting of SCOOP, the "waits-for" relation contains an association between between processors p and q iff some other processor has a lock on p and is requesting q.
- statically (conservative): arrange the processor tags into a partial order. When the text of the program indicates a lock is taken, verify that it is less than all the other locks that could have been taken at this point. The program text may require some annotations establishing which locks have already been taken.

These two schemes can be applied to Program 1. Reasoning using the dynamic scheme, we see that an instance of class DEADLOCK will lock its attributes x and y in some order when its routine f is called. In class MAIN, two instances d1 and d2 of DEADLOCK are initialized with two separate objects x and y, however

their order is reversed between the two instances. By executing `run`, the routine calls `d1.f` and `d2.f` are executed asynchronously, according to the semantics of calls on separate objects `d1` and `d2` outlined above.

As a result of executing `d1.f`, the call `g(x)` is invoked. As `x` is an argument to the routine `g`, the runtime locks `x` for the duration of the call, as prescribed by the semantics for controlled objects outlined above. In particular, `x` will still be locked when the call `h(y)` is invoked, requesting a lock on `y`. The concurrent execution of `d2.f` has an analogous locking behavior, but since `d1` and `d2` have opposite views of `x` and `y`, the locking order is reversed. Hence the calls may ultimately form a cyclical locking pattern, resulting in a deadlock.

To reason statically about the same sequence of calls, one notices that the order of calls can be conservatively approximated by examining the program text, and observing which routines subsequently call other routines. In the case of Program 1, we always know that calling the feature `f` will (for a general routine, may) require that the processor of `x` is locked, followed by `y`. This information can be used statically at the call sites of `d1.f` and `d2.f` to determine that their concurrent execution could lead to a deadlock state.

We have chosen to develop a static technique, as we believe that static techniques encourage the active construction of correct programs, whereas dynamic techniques cater more to a reactive development style.

3 A Formal Model of SCOOP Locking

Our approach uses a static detection scheme, requiring that the interfaces of a program be annotated. This includes routines and types of variables, where the annotations for variables follow the SCOOP-style very closely.

3.1 Annotation Language

At the class level, annotations of the following form are allowed:

$$\mathit{class_header} ::= \textbf{class}\ \mathit{ident}\ \big|\ \textbf{class}\ \mathit{ident}\ \text{<}p(,p)^{*}\text{>}$$

A class can thus be parameterized with the processor tags it is using. Consider Program 2 for example, an entity `d` based on class `DEADLOCK` uses processors with tags `p` and `q` for the roles of `xp` and `yp`. An instance is declared as follows:

```
d : DEADLOCK <p, q>
```

The preconditions of routines represent the required orderings of processors, expressed using the following syntax (note that we replace the non-strict ordering symbol by a strict ordering symbol in the program text to make it easier to type, however the interpretation should remain non-strict in all cases)

$$\mathit{req} ::= \epsilon\ \big|\ \textbf{require}\ p < p(,p < p)^{*}$$

```
  class DEADLOCK <xp,yp>

  feature
    x : separate <xp> S
5   y : separate <yp> S

    f
      require yp < xp
      lock xp
10    do
        g (x)
      end

15  g (a : separate <xp> S)
      require yp < xp
      do h (y) end

    h (b : separate <yp> S)
20    do end

    set (a_x : separate <xp> S;
         a_y : separate <yp> S)
      do
25      x := a_x
        y := a_y
      end
  end
```

Program 2: Annotated DEADLOCK class

For example, in line 16 the routine `g` is annotated to express that the processor `yp`, which will be locked as a result of the execution of the body of `g`, is below processor `xp`, which is locked by calling `g`.

In the interface of a routine we state the set of locks that may be taken temporarily during the execution of the routine body.

$$ens ::= \epsilon \;\Big|\; \textbf{lock}\, p(,p)^*$$

For example; in line 9 we state that a lock on `x`'s processor `xp` may be taken by executing the body of `f`, as the call `g(x)` will lock this processor. Note that small changes to the program (re-nesting function calls, for instance) may require the ordering specifications to be modified accordingly.

In Program 2 we require that `yp < xp` in feature `g`. Due to the construction of the two deadlock variables `d1` and `d2` in Program 1, we know that the two classes are instantiated with conflicting requirements: one requires that `yp < xp` and the other will then necessitate `xp < yp`. Since these cannot be mutually satisfied, it is impossible to annotate `MAIN` from Program 1 such that it can satisfy the well-formedness predicate.

3.2 SCOOP Program Model

Complementing the program annotations, we provide a formalization of SCOOP programs based on the computational model described in Section 2.1. We focus on routines as the basic units of programs, as it is at routine invocation that locks are taken, and at routine return where lock reservations are given up. We disregard classes and class-level processor annotations as they introduce unnecessary complexity in the representation; the formalization could however be extended to include them.

Assume to have a set of (routine) names *Name*. We consider a *program* $\mathcal{P}$ to be a mapping of names to routines

$$\mathcal{P} \in \mathit{Program} = \mathit{Name} \rightarrow \mathit{Routine}$$

where an (unnamed) routine *rtn* is of the form

$$\mathit{rtn} \in \mathit{Routine} = \wp(\mathit{Tag} \times \mathit{Tag}) \times \mathit{Tag}^* \times \wp(\mathit{Tag}) \times \mathit{Tag} \times \mathit{Expr}$$

and we refer to its components using the following notation:

$$rtn = (rtn_{\leq}, rtn_{\text{args}}, rtn_{\text{locks}}, rtn_{\text{res}}, rtn_{\text{body}})$$

The component $rtn_{\leq}$, corresponding to programmer provided **require** annotations as in Section 3.1, is a relation on processor tags, describing the partial order on processors required by the routine. The component rtn_{args} is the sequence of formal arguments of the routine. The set of locks that may be taken as the result of executing the body of the routine is given by rtn_{locks}, corresponding to **lock**; this is the other programmer-provided annotation. The component rtn_{res} specifies whether the routine returns a result or not, and rtn_{body} is the body of the routine (an expression).

An expression e is constructed from the following syntax

$$e \ ::= \ [p] \ \mid \ \mathsf{skip} \ \mid \ \mathsf{create}(p) \ \mid \ e \cdot f(\tilde{e}) \ \mid \ e; e \ \mid \ \mathsf{waitfor}(p) \ \mid \ \mathsf{unlock}$$

where $p \in \mathit{Tag}$ is a processor tag and $f \in \mathit{Name}$ is a name. We write $\tilde{e}$ to abbreviate a sequence of expressions $e_1, \ldots, e_n$, and similarly $\tilde{p}$ for a sequence of processors. We sometimes treat these sequences as sets, i.e. $\tilde{e} = \bigcup_{i=1,\ldots,n}\{e_i\}$. We assume that processor tags can be inferred from an expression e using a mapping $tag_{\mathcal{P}} : \mathit{Expr} \rightarrow \mathit{Tag}$; we use $tags_{\mathcal{P}}$ on sequences of expressions.

The syntax elements have the following intuitive meaning. A value on a processor p is represented as $[p]$, abstracting away the actual value. Since the actual value is discarded, assignments in a program text are transformed into only the right-hand side, as it may contain some call (and thus locking). If in a sequence $\tilde{e} = e_1, \ldots, e_n$ all expressions are fully evaluated (i.e. $e_i = [p_i]$ for $i = 1, \ldots, n$), we use the notation $[\tilde{e}]$. The expression skip has no effect. A new processor with tag p is created using $\mathsf{create}(p)$. Calling a routine f on a target t, with a list of arguments $\tilde{a}$ is represented by $t \cdot f(\tilde{a})$. Sequencing of expressions is written as $e_1; e_2$. The remaining two syntax elements $\mathsf{waitfor}(p)$ and unlock do not represent program syntax, but are required for the purpose of modeling the waiting and locking behavior of the runtime system. Waiting on a processor with tag p is expressed as $\mathsf{waitfor}(p)$. The expression unlock represents unlocking of the set of processors that has been taken as a result of the matching routine call.

3.3 Locking Semantics

Given the formalization of SCOOP programs, we can proceed to formally defining the part of the program semantics that is critical for reasoning about deadlock. Rather than enabling us to reason about full program correctness, the following rewrite rules embody the behavior of requesting, taking and releasing locks in a SCOOP program.

At runtime, a program $\mathcal{P}$ gives rise to a *process* P which is described by the following syntax:

$$P \ ::= \ p :: e \ \mid \ P \mid P$$

A process is therefore either an expression e located at a processor with tag p, or a parallel composition of processes. The idea is that a program starts with

the initial call f_0 on an initial processor p_0 as $p_0 :: f_0$, and will give rise to more parallel threads (as the result of create) as execution proceeds. A structural equivalence $\equiv$ over processes specifies the commutativity and associativity of the $|$ operator; the formal definition of $\equiv$ is standard and omitted from this presentation. We assume that processor tags are unique within processes, i.e. there cannot be a process $P \equiv p :: e \mid q :: e' \mid Q$ such that $p = q$. This property is preserved by process creation.

Processes are operating on a state representing locks and requests only. Formally, we define a *lock state* L as a pair of mappings (L_l, L_r) of the following type:

$$L \in \mathit{LockState} = (\mathit{Tag} \to (\wp(\mathit{Tag}))^*) \times (\mathit{Tag} \to \wp(\mathit{Tag}))$$

Here, L_l is a mapping from a processor (tag) to a stack of sets of processors, representing the processors it currently locks. Although a set of locks would suffice here, having a stack of sets allows for a greater correspondence with the intuition that lock-taking in SCOOP closely follows the call-stack. We define the domain of L as the union of the domains of its components, $\mathit{dom}(L) = \mathit{dom}(L_l) \cup \mathit{dom}(L_r)$. We use the notation $L_l[p \mapsto T]$ for updates, such that the resulting mapping returns T at point p of its domain and is unchanged otherwise. We write $T : \mathit{lcks}$ for a stack obtained by pushing a set of processor tags T on a stack lcks. We write $\bigcup \mathit{lcks}$ for flattening the stack into one set, i.e. if $\mathit{lcks} = T_1, \ldots, T_n$ then $\bigcup \mathit{lcks} = \bigcup_{i=1,\ldots,n} T_i$. L_r is a mapping from a processor to the set of processors it requests locks for. The requested processors are tracked to align our model with the Coffman treatment of when deadlock occurs.

The locking semantics specifies rewrite rules over processes and lock states in the style of a structural operational semantics with transitions of the form:

$$\mathcal{P} \vdash (P, L) \to (P', L')$$

This means that, given a program $\mathcal{P}$ which provides meaning to names of routines occurring in processes, the process P evolves in one step to P' and transforms locking state L to L'.

With this information, we can now look to the rules contained in Table 1 for the definition of the locking semantics. The creation of a new processor q by a processor p gives rise to a new parallel process located at q. If the processor already exists, then this has no effect. These behaviours can be see in the CREATE$_1$ and CREATE$_2$ rules.

The rule SEQ allows one step to be performed on the left side of a sequential composition, and SKIP carries its intuitive meaning. For routine target and argument evaluation: EVAL-TRG and EVAL-ARG enforce that targets are fully evaluated before arguments are evaluated. In EVAL-ARG, the arrow $\twoheadrightarrow$ represents performing a single rewrite step on a sequence of expressions. To reorder the constituent processes of a program during rewriting, the EQUIV rule is available.

Once the target and arguments of a call are both fully evaluated, the call can be invoked. In the case where the call has no result, CALL-NORES moves the call to the target processor, to be executed after the current tasks of the target processor; the caller proceeds without waiting. Recall that we use the

Table 1. SCOOP Rewrite Rules

CREATE$_1$
$$\frac{p \neq q \qquad Q \not\equiv q :: e \mid Q'}{\mathcal{P} \vdash (p :: \mathsf{create}(q) \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid q :: \mathsf{skip} \mid Q, L)}$$

CREATE$_2$
$$\frac{p = q \ \vee\ Q \equiv q :: e \mid Q'}{\mathcal{P} \vdash (p :: \mathsf{create}(q) \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid Q, L)}$$

EVAL-TRG
$$\frac{\mathcal{P} \vdash (p :: t \mid Q, L) \rightarrow (p :: t' \mid Q', L')}{\mathcal{P} \vdash (p :: t \cdot f(\tilde{a}) \mid Q, L) \rightarrow (p :: t' \cdot f(\tilde{a}) \mid Q', L')}$$

EVAL-ARG
$$\frac{\mathcal{P} \vdash (p :: \tilde{a} \mid Q, L) \twoheadrightarrow (p :: \tilde{a}' \mid Q', L')}{\mathcal{P} \vdash (p :: [q] \cdot f(\tilde{a}) \mid Q, L) \rightarrow (p :: [q] \cdot f(\tilde{a}') \mid Q', L')}$$

SKIP
$$\frac{}{\mathcal{P} \vdash (p :: \mathsf{skip}; e \mid Q, L) \rightarrow (p :: e \mid Q, L)}$$

SEQ
$$\frac{\mathcal{P} \vdash (p :: e_1 \mid Q, L) \rightarrow (p :: e_1' \mid Q', L')}{\mathcal{P} \vdash (p :: e_1; e_2 \mid Q, L) \rightarrow (p :: e_1'; e_2 \mid Q', L')}$$

EQUIV
$$\frac{P \equiv Q \qquad Q' \equiv P' \qquad \mathcal{P} \vdash (Q, L) \rightarrow (Q', L')}{\mathcal{P} \vdash (P, L) \rightarrow (P', L')}$$

CALL-NORES
$$\frac{\mathcal{P}(f)_{result} = \text{None} \qquad Q \equiv q :: e \mid Q' \qquad q \in \bigcup L_l(p)}{\mathcal{P} \vdash (p :: [q] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid q :: e; [q] \cdot f([\tilde{a}]) \mid Q', L)}$$

CALL-RES
$$\frac{\mathcal{P}(f)_{result} = \text{Some}(v) \qquad Q \equiv q :: e \mid Q' \qquad q \in \bigcup L_l(p)}{\mathcal{P} \vdash (p :: [q] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow (p :: \mathsf{waitfor}(q) \mid q :: e; [q] \cdot f([\tilde{a}]) \mid Q', L)}$$

REQ-LCK
$$\frac{need = \tilde{a} - (\bigcup L_l(p) \cup \{p\}) \qquad L_r' = L_r[p \mapsto need] \qquad L_l' = L_l}{\mathcal{P} \vdash (p :: [p] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow (p :: [p] \cdot f([\tilde{a}]) \mid Q, L')}$$

LOCK
$$\frac{L_r' = L_r[p \mapsto \emptyset] \qquad \tilde{a}' = \mathcal{P}(f)_{arg} \qquad \left(\bigcup_{x \in dom(L)} \bigcup L_l(x)\right) \cap L_r(p) = \emptyset \qquad L_l' = L_l[p \mapsto L_r(p) : L_l(p)]}{\mathcal{P} \vdash (p :: [p] \cdot f([\tilde{a}]) \mid Q, L) \rightarrow (p :: \mathcal{P}(f)_{body}[\tilde{a}/\tilde{a}']; \mathsf{unlock} \mid Q', L')}$$

RET
$$\frac{Q \not\equiv q :: \mathsf{waitfor}(p) \mid Q' \qquad L_r' = L_r \qquad L_l(p) = T : lcks \qquad L_l' = L_l[p \mapsto lcks]}{\mathcal{P} \vdash (p :: [v]; \mathsf{unlock} \mid Q, L) \rightarrow (p :: [v] \mid Q, L')}$$

RET-WAIT
$$\frac{Q \equiv q :: \mathsf{waitfor}(p) \mid Q' \qquad L_r' = L_r \qquad L_l(p) = T : lcks \qquad L_l' = L_l[p \mapsto lcks]}{\mathcal{P} \vdash (p :: [v]; \mathsf{unlock} \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid q :: [v] \mid Q', L')}$$

UNLOCK
$$\frac{L_l(p) = T : lcks \qquad L_r' = L_r \qquad L_l' = L_l[p \mapsto lcks]}{\mathcal{P} \vdash (p :: \mathsf{unlock} \mid Q, L) \rightarrow (p :: \mathsf{skip} \mid Q, L')}$$

notation $[\tilde{a}]$ to describe a fully evaluated sequence of expressions, and $\bigcup L_l(p)$ for flattening the stack of locks $L_l(p)$. To make a call on a separate target, we require the processor p to hold a lock on the target processor q with the condition $q \in \bigcup L_l(p)$. When the call has a result, the dispatching processor must wait on the result from the target processor, as in CALL-RES.

Upon a call arriving on its target processor, the required locks must be requested, specified in REQ-LCK. We only request the locks we do not already hold, which are collected in the set *need*; the local processor is never *need*ed. Once the requests have been made, they are transferred to the lock set (LOCK) of the processor when no other processor has any of the locks. Then the body of the routine is scheduled for evaluation, followed by a request to unlock all initially requested locks after the execution of the body has been completed. Here we use the notation $\mathcal{P}(f)_{body}[\tilde{a}/\tilde{a}']$ to substitute the sequence of actual arguments $\tilde{a}$ for the formal arguments $\tilde{a}' = \mathcal{P}(f)_{arg}$ within the body $\mathcal{P}(f)_{body}$ of routine f. In the previous two rules, the sequence of values $[\tilde{a}] = [p_1], \ldots, [p_n]$ is reinterpreted in set computations as a set of processors, i.e. $\tilde{a} = \bigcup_{i=1,\ldots,n}\{p_i\}$.

The $\mathsf{waitfor}$ primitive allows a value that has been computed on a target processor q to be transferred to the processor p that is waiting for it (compare

rule CALL-RES). As the returning of the value also completes a call, the locks that have been taken as a result of the call are also released (L'_l is obtained from L_l by popping one element off the stack). Two rules are required to return values to callers (RET and RET-WAIT), one which would be the result of a non-separate call (no waitfor), and one which has an accompanying waitfor on another processor:

For the case where a call has been completed but no result is returned (compare rule CALL-NORES) there may be no value $[v]$ sitting before the unlock, so an analogous rule for unlocking is needed with UNLOCK.

Example 1. To illustrate the use of the rewrite rules, we apply them to Program 1. System execution starts with a call `make` on an initial processor p. We show an execution step of the body of `make`, demonstrating an application of rule CREATE$_1$ on the instruction **`create`** `x`:

$$(p :: \mathsf{create}(q); e \mid Q, L) \rightarrow (p :: e \mid q :: \mathsf{skip} \mid Q, L)$$

Here we assume that the processor tag of the local variable `x` is q and can be obtained with a mapping $tag : Name \rightarrow Tag$.

The other **`create`**-statements will give rise to more concurrent processes. Finally, the routine `run` is called, and we assume that $tag(\mathtt{d1}) = r_1$ and $tag(\mathtt{d2}) = r_2$ to get the following derivation.

$$\begin{aligned}
&(p :: [p] \cdot \mathtt{run}([r_1], [r_2]); e' \mid r_2 :: \mathsf{skip} \mid r_1 :: \mathsf{skip} \mid Q', L) \rightarrow \\
&(p :: [p] \cdot \mathtt{run}([r_1], [r_2]); e' \mid Q'', (L_l, L_r[p \mapsto \{r_1, r_2\}])) \rightarrow \\
&(p :: [r_1] \cdot \mathtt{f}; [r_2] \cdot \mathtt{f}; \mathsf{unlock} \mid Q'', (L_l[p \mapsto \{r_1, r_2\} : L_l(p)], L'_r[p \mapsto \emptyset])) \rightarrow \\
&(p :: [r_2] \cdot \mathtt{f}; \mathsf{unlock} \mid r_2 :: \mathsf{skip} \mid r_1 :: \mathsf{skip}; [r_1] \cdot \mathtt{f} \mid Q', L'')
\end{aligned}$$

Here, the first step is due to rule REQ-LCK and shows that the processors of `d1` and `d2` are added to the request set of p. The second step is then according to rule TAKE-LCK, and shows that the requested locks (which are available) are taken by pushing them on p's stack of locks. The last step is an application of rule CALL-NORES and shows how an asynchronous call is transferred to its handling processor. Applications of rules SKIP and SEQ are omitted for brevity.

4 Deadlock Prevention Scheme

In this section we present a scheme for deadlock prevention, based on annotations in Section 3.1. We define well-formedness of annotated programs. We prove that well-formed programs cannot deadlock, based on our formalization of the locking semantics in Section 3.3.

4.1 Well-Formed Programs

The scheme for ensuring that a program is well-formed ensures that there exists, for each routine, a consistent processor ordering (through $rtn_{\leq}$). Additionally, it ensures that locks are declared (rtn_{locks}) properly, and within the scope of these

declared locks the callee's locks (rtn'_{locks} instantiated by its arguments) do not lose any of the knowledge that the declared locks are held. The well-formedness property of a program can be formally stated as a predicate:

$$wfProgram_{\mathcal{P}} = \forall rtn \in range(\mathcal{P}).\ wfRoutine_{\mathcal{P}}(rtn)$$

A well-formed routine must ensure that it's interface is well-formed (first clause) and also that the routine body is consistent with the interface (second clause):

$$wfRoutine_{\mathcal{P}}(rtn) = isOrder(rtn_{\leq}) \wedge wfExpr_{\mathcal{P}}(rtn_{\leq}, rtn_{locks}, rtn_{body})$$

The definition of a well formed expression allows neither waitfor nor unlock in the program text, these are only inserted at runtime by the rewrite process. The well-formedness of expressions is thus given by the following definition:

$$\begin{aligned}
wfExpr_{\mathcal{P}}(\leq, lks, [p]) &= True \\
wfExpr_{\mathcal{P}}(\leq, lks, \mathsf{skip}) &= True \\
wfExpr_{\mathcal{P}}(\leq, lks, \mathsf{create}(p)) &= True \\
wfExpr_{\mathcal{P}}(\leq, lks, e_1; e_2) &= wfExpr_{\mathcal{P}}(\leq, lks, e_1) \wedge wfExpr_{\mathcal{P}}(\leq, lks, e_2) \\
wfExpr_{\mathcal{P}}(\leq, lks, t \cdot f(\tilde{a})) &= inst_{\leq} \subseteq \leq \wedge wfLevels_{\mathcal{P}}(\leq, inst, lks, \tilde{a}) \wedge \\
&\quad \forall a \in \tilde{a}.\ \ tag_{\mathcal{P}}(a) \leq tag_{\mathcal{P}}(t) \wedge wfExpr_{\mathcal{P}}(\leq, lks, a) \\
&\qquad \text{where } \ inst = \mathcal{P}(f)[\tilde{a}/\mathcal{P}(f)_{args}]
\end{aligned}$$

We treat the cases of values, skip, create, and sequencing with less detail here: they are either immediately well-formed or are well-formed based on a trivial recursion. The first clause of the call-case of *wfExpr* states that the instantiated routine interface must have its order consistent with the context-order ($\leq$). The second clause states that the lock-level is respected. The third clause states that each argument is a well-formed expression, and its processor is less than the target of the call.

$$wfLevels_{\mathcal{P}}(\leq, inst, lks, \tilde{a}) = ((inst_{locks} \times lks) \ \subseteq \leq) \wedge (tags_{\mathcal{P}}(\tilde{a}) \subseteq inst_{locks})$$

The first clause of *wfLevels* has all associations between the declared locks of the call and the context locks being also in the order relation. In other words, this states that each declared lock of a call must be less than all of the context-locks, so that we only lock "down" the partial order. Since a routine may have no arguments and still lock some processors in its body we compare context-locks against the **lock** clause, and not the arguments. The second clause states that if a routine does have arguments, then these arguments must be a subset of the **lock** clause, for consistency.

Example 2. For the Program 2, we show the evaluation the predicate *wfExpr* on the call of the routine `g` in the body of routine `f`. To make the example more varied, assume that the argument $[xp]$ and the corresponding lock of `g` are replaced by $[zp]$.

We let $ord = \{(yp, xp), (xp, t), (xp, xp), (yp, yp), (t, t)\}$. As $(xp, t) \in ord$ and $inst_{\leq} = ord$, the predicate is satisfied:

$$\begin{aligned} wfExpr(ord, \{xp\}, [t] \cdot g([xp])) = {} & (\forall a \in [xp].\ (tag_{\mathcal{P}}(a), t) \in ord \wedge \\ & \quad wfExpr_{\mathcal{P}}(ord, \{xp\}, [xp])) \\ & \wedge\ inst_{\leq} \subseteq ord \\ & \wedge\ wfLevels(\{(yp, xp), (xp, t)\}, inst, \{xp\}, \{xp\}) \end{aligned}$$

Here we use that

$$\begin{aligned} \mathcal{P}(g) &= (\{(yp, zp), (zp, t), (zp, zp), (yp, yp), (t, t)\}, [zp], \{zp\}, None) \\ inst &= \mathcal{P}(g)[[xp]/[zp]] = (ord, [xp], \{xp\}, None) \end{aligned}$$

and that the predicate *wfLevels* is satisfied because values are well-formed, and (xp, xp) is in the order (reflexivity).

$$wfLevels(ord, inst, \{xp\}, [xp]) = (\{xp\} \times \{xp\}) \subseteq ord \wedge tags_{\mathcal{P}}([xp]) \subseteq \{xp\}$$

4.2 Deadlock Freedom

Intuitively, our scheme ensures that there exists a global ordering for every well-formed program, and also that during execution of this program each processor obeys an order in which to take locks. Deadlock-freedom follows from the fact that the acyclicity of the locking state is preserved under any execution step.

To formalize these ideas, we build on notion of a locking graph from [5]. We do not directly show that the rewriting of the operational semantics can not get "stuck" due to lock requests, although this property follows from the locking graph formalization. Translated to our setting, a locking graph has processors (resources) as nodes. There is an edge (p, q) in the graph if some process has locked processor p while requesting processor q. A locking-state L induces a locking-graph relation $graph(L)$ as follows, where $Id_{dom(L)}$ is the identity relation on processors in the domain of L:

$$graph(L) = Id_{dom(L)} \cup \left(\bigcup_{p \in dom(L)} L_l(p) \times L_r(p) \right)$$

The information provided by the lock state L, and associated locking-graph, is not rich enough to prove the properties that will be needed. We therefore introduce two new concepts: a lock-barrier $L_b : Tag \to (\wp(Tag))^*$ and a runtime ordering $L_{\leq} \in \wp(Tag \times Tag)$. The lock barrier represents the set of upper bounds on the locks we are allowed to request. The runtime ordering is the ordering which is built up during execution. For the sake of the proof, the locking semantics has to be instrumented with these concepts. The minimal additions to the semantics are shown in Table 2. For our approach, all locks taken have to stay below the current locking barrier at any time, and the runtime ordering is the order that is built at runtime as a result of the order annotations.

Table 2. Instrumented rules

LOCK
$$\frac{L'_b(p) = (f_{locks}[\tilde{a}/\tilde{a}']) : L_b(p) \quad L'_{\leq} = (L_{\leq} \cup (f_{\leq}[\tilde{a}/\tilde{a}']))^* \quad \ldots}{\ldots}$$

RET
$$\frac{L_b(p) = b : L'_b(p) \quad \ldots}{\ldots}$$

RET-WAIT
$$\frac{L_b(p) = b : L'_b(p) \quad \ldots}{\ldots}$$

UNLOCK
$$\frac{L_b(p) = b : L'_b(p) \quad \ldots}{\ldots}$$

We prove that the following predicate, *sound*, is invariant under execution. The predicate states that the runtime ordering $L_{\leq}$ is indeed a partial order, that the locking barrier is respected, and that the locking graph is acyclic.

$$\begin{aligned} sound(L) = {} & isOrder(L_{\leq}) & (1)\\ & \wedge \forall p \in dom(L).\ top(L_b(p)) \times \bigcup L_l(p) \subseteq L_{\leq} & (2)\\ & \wedge graph(L) \subseteq L_{\leq}^{-1} & (3)\end{aligned}$$

Here, $L_{\leq}^{-1}$ denotes the converse of the relation $L_{\leq}$, and *top* denotes the first element of a sequence.

Theorem 1. *Given a well-formed program $\mathcal{P}$ and an instrumented rewrite rule $\mathcal{P} \vdash (P, L) \to (P', L')$, $sound(L)$ implies $sound(L')$.*

Briefly, the third clause is of primary concern; if the locking-graph ($graph(L)$) is a subset of an order, then it must be acyclic. Since $L_{\leq}$ is an order, thus acyclic, so is its inverse.

The initial two clauses support this goal, with the first establishing that as the program executes the relation that is specified piece-wise in the routine annotations is indeed an order. This fact follows from the definition of *wfRoutine* and the instantiation of the routines in the first clause of the call-case of *wfExpr*.

The second clause of *sound* states that the new upper-bound on locks is below all other locks that have already been acquired by the processor p. The proof of this property is garnered from the Cartesian product in the *wfLevels* predicate, which imposes that when locks are taken, they are statically less than every lock taken by the surrounding procedure. When function calls are nested, these transitively combine to ensure that locks requested by a processor p are less than all other locks currently held by that processor. Since we know that the locking scheme preserves the order relation, it must also preserve the inverse order relation, which is the essential property desired to prove the third clause.

4.3 Usage and Tool Support

We have implemented the static checking of our scheme in a prototype tool, written in Haskell [9]. Using this tool we successfully verified that a simple web server is deadlock-free, a portion of which can be seen in Program 3.

To reduce the annotation burden, we have also implemented a simple annotation inference algorithm. The annotations shown in Program 2 can be automatically inferred using the tool. The simple inference scheme automatically identifies **separate** class attributes with processor tags and lifts the tags to the

```
  db : separate <d> DATABASE

  req (sock : separate <s> NET_SOCK)
    require d < s
5   local
      last    : STRING
      http_req : HTTP_REQUEST
    do
      create http_req.make ()
10
      from read_line (sock)
      until last.is_equal (cr)
      loop
        http_req.add_field (last)
15      read_line (sock)
      end
      update_database (db, http_req)
      process_request (http_req)
    end
```

Program 3: HTTP request processing

class header. It also propagates **lock** and **require** clauses appropriately, based on calls within the body of a routine. For example, at a call-site, the **require** clause of the call would be automatically appended to the containing routine's **require** clause; a similar approach is taken for the **lock** clause. This typically makes the manual annotation burden light.

5 Related Work

The problem of describing, detecting, and preventing deadlocks in concurrent systems has spawned research based on a variety of approaches. Necessary conditions for a deadlock to occur have been described in a seminal work by Coffman et al. [5]. *Dynamic techniques* can be used to detect deadlocks, e.g. using techniques such as those presented by Bensalem et al. [2]. The fundamental approach in this work is to instrument the program and use this runtime locking information to detect locking cycles. The benefit is that this technique can be less conservative than our approach, but it is based on actual program traces, and the results are, therefore, not sound.

Static techniques rely on programmer annotations to indicate a partial order among the program's locks, and statically check whether this order is abided by; this general idea is also the basis of our approach. Korty [13] proposed a Lint-like tool for detecting deadlocks in programs with semaphores, however without soundness guarantees. Extended static checking for Modula-3 [6] and Java [7] uses program specifications in the style of Eiffel [15], from which verification conditions are generated and checked with an automatic theorem prover. Warnings are provided for various program errors, including deadlock. Being based on Eiffel-style specifications, annotations in this approach are similar to our scheme. However, no soundness guarantees are given whereas we guarantee deadlock-freedom for well-formed programs. Jacobs et al. [10] also generate verification conditions for annotated programs, and guarantee deadlock-freedom for programs verified with a static checker. In contrast to our work, they use a programming model for Java-like languages which is very different from SCOOP, and do not provide a rigorous formal locking semantics.

A number of static approaches to deadlock prevention are based on *type systems*, in particular using ownership types [4]. Boyapati et al. [3] have introduced the ability, as in our approach, to create a directed acyclic graph, well-order, or

tree to represent the underlying partial order. In contrast to this approach, our scheme makes it possible to declare locking orders in a routine-local manner, which allows for a finer-grained modularity.

Our work is distinguished from the above approaches in that it has a higher-level concurrency model, not based on traditional threads, and thus has a coarser-grained locking model.

Using a model similar to SCOOP, Kerfoot et al. [11] use types to ensure deadlock freedom for active objects [14]. Ownership types impose a hierarchy on active objects, but the variety of ownership-structures that are permitted are limited. Only trees are allowed, where our approach can support a general directed acyclic graph. Ostroff et al. [17] develop a partial operational semantics for SCOOP, and consider liveness properties of programs in the context of model checking. While the approach can detect deadlocks, it is not modular, thus does not scale to large programs. Kobayashi [12] gives π-calculus a type system that is able to infer and verify deadlock properties about a program. It gives a versatile approach that is even able to reason about recursive processes. However, our work targets a new model of computation that is more immediately amenable to traditional imperative programming.

6 Conclusion

In this paper we have presented a static technique for deadlock prevention in SCOOP, an object-oriented programming model for concurrency. We found that the model supports well reasoning about deadlock, as lock acquisition and release are related to routine invocation and return. This allows the annotations to be attached to the interface of routines, facilitating modular (per-routine) proofs of correctness. This aspect is essential in practice as it is easier to reason about deadlock when it is assured that local changes will not affect the overall result. An implementation of the scheme is available, and has been successfully applied to the example of a web server written in SCOOP.

Adding a deadlock prevention technique for SCOOP removes a critical deficiency of this particular model, but the results also provide important general lessons learned. While sound and scalable programming models for concurrency are overdue, the divide between formally driven language developments (such as process calculi) and concurrent programming language design still seems to be large. This work showcases how one may bridge this gap by using formal reasoning to derive techniques that can be applied to practical programming languages.

In future work we will investigate the possibility of statically avoiding deadlock by creating some objects on the same processor when not rejected by other constraints, expanding on the annotation inference techniques. Work on the semantic foundations of the programming model provides also many avenues for future research. The distributed nature apparent in the semantics can give important insights into extending the programming model for distribution. Also, variants of the semantics can be studied, for example to provide insights about possible performance improvements.

Acknowledgments. This work is part of the SCOOP project at ETH Zurich, which has benefitted from grants from the Hasler Foundation, the Swiss National Foundation, Microsoft (Multicore award) and ETH (ETHIIRA).

References

1. Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races. In: Proc. OOPSLA 2000, pp. 382–400. ACM, New York (2000)
2. Bensalem, S., Fernandez, J., Havelund, K., Mounier, L.: Confirmation of deadlock potentials detected by runtime analysis. In: PADTAD 2006, pp. 41–50. ACM, New York (2006)
3. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: preventing data races and deadlocks. In: Proc. OOPSLA 2002, pp. 211–230. ACM, New York (2002)
4. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for flexible alias protection. ACM SIGPLAN Notices 33(10), 48–64 (1998)
5. Coffman, E.G., Elphick, M., Shoshani, A.: System deadlocks. ACM Computing Surveys 3(2), 67–78 (1971)
6. Detlefs, D.L., Leino, R., Nelson, G., Saxe, J.B.: Extended static checking. Technical Report 159, Compaq SRC (1998)
7. Flanagan, C., Leino, R., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI 2002, pp. 234–245. ACM, New York (2002)
8. Hoare, C.A.R.: Monitors: an operating system structuring concept. Communications of the ACM 17(10), 549–557 (1974)
9. SCOOP homepage (2010), http://scoop.origo.ethz.ch/
10. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A statically verifiable programming model for concurrent object-oriented programs. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 420–439. Springer, Heidelberg (2006)
11. Kerfoot, E., McKeever, S., Torshizi, F.: Deadlock freedom through object ownership. In: Proc. IWACO 2009, pp. 1–8. ACM, New York (2009)
12. Kobayashi, N.: A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 233–247. Springer, Heidelberg (2006)
13. Korty, J.A.: Sema: A lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In: USENIX Winter Technical Conference (1989)
14. Lavender, R.G., Schmidt, D.C.: Active object: an object behavioral pattern for concurrent programming. In: Pattern Languages of Program Design, pp. 483–499. Addison-Wesley, Reading (1996)
15. Meyer, B.: Object-Oriented Software Construction, 2nd edn. Prentice-Hall, Englewood Cliffs (1997)
16. Nienaltowski, P.: Practical framework for contract-based concurrent object-oriented programming. PhD thesis, ETH Zurich (2007)
17. Ostroff, J.S., Torshizi, F., Huang, H.F., Schoeller, B.: Beyond contracts for concurrency. Formal Aspects of Computing 21(4), 319–346 (2009)
18. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15(4), 391–411 (1997)
19. Torshizi, F., Ostroff, J.S., Paige, R.F., Chechik, M.: The SCOOP concurrency model in Java-like languages. In: Proc. CPA 2009. IOS Press, Amsterdam (2009)

Model-Driven Protocol Design Based on Component Oriented Modeling

Prabhu Shankar Kaliappan, Hartmut König, and Sebastian Schmerl

Department of Computer Science
Brandenburg University of Technology Cottbus
P.O. Box 10 13 44, 03013 Cottbus, Germany
{psk,koenig,sbs}@informatik.tu-cottbus.de

Abstract. Due to new emerging areas in the communication field there is a constant need for the design of novel communication protocols. This demands techniques for a rapid and efficient protocol design and development. Systematic protocol designs using formal description techniques (FDTs), such as SDL, LOTOS, etc., have proven a successful way to develop correct protocols. FDTs enforce, however, a semantic-oriented description which makes it difficult to reuse parts of the specification of other FDTs. A general-purpose modeling language like the UML may help to easily bridge between different description techniques. In contrast to the standardized FDTs, UML lacks a formal semantics. A model-driven protocol design, which aims at supporting the reuse of designs, makes only sense, when the designs of basic protocol mechanisms fit in reusable design patterns or components with a formally defined semantics. In this paper, we propose a component based protocol development approach with UML. Typical structures and behaviors of protocols are pre-defined as components using UML diagrams. The semantics of the UML diagrams is formally defined using the compositional Temporal Logic of Actions (cTLA). Based on this formalization, transformation into other presentations, e.g. PROMELA for verification, are supported. We demonstrate the approach for an example transfer protocol.

Keywords: communications protocols, UML based protocol development, component based design, formal semantics.

1 Motivation

Reducing the development time and increasing the quality of a product are crucial factors in software development. Many techniques and approaches have been introduced in day-to-day life for the development of a product, but selecting the appropriate technique for a given problem is decisive for the quality and efficacy of the development. This also applies to the development of communication protocols. There is a constant need for the design of new protocols, since new areas emerge in the communication field. The development of protocols from scratch is usually a lengthy process, error-prone, and time-consuming. On the other hand, protocols often

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 613–629, 2010.
© Springer-Verlag Berlin Heidelberg 2010

apply similar basic mechanisms, such as flow control, error control, connection maintenance, etc. The protocol development can be considerably accelerated by the reuse of such mechanisms.

There has been a long tradition in systematic protocol developments using formal description techniques (FDTs), such as the *Specification and Description Language* (SDL) [1] or the *Language of Temporal Ordering Specification* (LOTOS) [2], etc. The deployment of formal description techniques has proven a successful way to design, verify, and test protocols because of their formal semantics basis. FDTs enforce, however, a semantic-oriented description, e.g. an agent-oriented description in SDL or a process oriented specification in LOTOS. The FDT based protocol development assumes a familiarity of the protocol engineer with the applied description technique. In practice, the use of FDTs is limited and mostly users specialize on one language. Verification and test case generation also relate to the applied description technique. Moreover, the various FDTs support due to their defined semantics only in part typically intuitive protocol modeling concepts, such as a representation of the message exchange between entities. This can be overridden by introducing *Unified Modeling Language* (UML) into the protocol design and deployment process. UML provides all means for the modeling of communication protocols and services. In contrast to the FDTs, UML supports with its various diagrams multiple views to describe (visualize) the different aspects of the protocol behavior during design, thus enabling a model-driven protocol development. Besides, UML has found a wide acceptance in academia and industries, much broader than the FDTs did. However, UML has not widely applied in the protocol area because it lacks a formal semantics.

A model-driven protocol design, which aims at supporting the reuse of designs, makes only sense if the designs of basic protocol mechanisms fit in reusable design patterns or components which can be composed with other components to a new design. Thus, intuitive protocol modeling patterns or components can be used as a basis for transformations into other presentations, e.g. other FDTs, executable code, or automata presentations for test case generation. Therefore, a component approach is only applicable if a formal semantics is allocated to these components. The correctness of the components and their composition has to be proved formally based on the given formal semantics. If this is given, UML based design components can be used for the design of new protocols, and can serve as basis for transformations into other presentations. In this paper, we present an approach for a component oriented protocol design and development process using UML with a formal semantics basis. Typical protocol procedures are pre-defined in UML diagrams for the reuse in various designs. The semantics of the UML diagrams is formally defined using the compositional Temporal Logic of Actions (cTLA) [8]. Based on this formalization, transformations into various representations, e.g. for verification and code generation, are supported. Thus, we are able to set up a similar development process as with standardized FDTs.

The remainder of the paper is organized as follows. In Section 2, we discuss related approaches. Thereafter in Section 3, we introduce the core of our component approach and its formal semantics definition. In Section 4, we present the component

compositions through an example data transfer protocol. We conclude the paper with a short summary and address next research steps.

2 Related Work

The use of design patterns and components has already been proposed and applied in various approaches for protocol and service design [3, 4], also in the context with formal description techniques. So Gotzhein et al. [5, 6] developed an approach for developing protocol specification in SDL using *SDL design patterns*. For the design, the requirements are captured by means of UML object diagrams as an *analysis model*. It is used as communication service architecture to identify the service users and the provider including the relations between them. The collaboration between the identified systems or objects is described in a Message Sequence Chart (MSC). The SDL design pattern approach is illustrated for the *Initiator-Responder* (InRes) protocol as case study. The approach uses several heuristic steps to derive the patterns. Hence, the protocol development is restricted within these steps for an unskilled protocol developer. The verification of independent SDL design patterns is not illustrated. Instead, the design is verified after composition, but not incrementally.

Byun et al. [7] propose with a pattern based development methodology for communication protocols, a similar approach as the SDL design patterns. They specify the patterns through *communicating extended finite state machines* (CEFSM) and exploit SDL as an implementation language. CEFSM applies a similar notation as used for state charts. The protocol is developed by applying only patterns, i.e. a pattern for connection, a pattern for data transfer and a merge pattern for combining various patterns. The verification is performed by mapping individual patterns onto the *Process Meta Language* (PROMELA) for SPIN model checking [14]. Here, the consistency among the used patterns and the SDL specification is not verified.

Herrmann et al. [8] introduced a framework for modeling transfer protocols using the specification technique *compositional Temporal Logic of Actions* (cTLA) as specification patterns. cTLA supports a modular definition of generic process types and the composition of process systems. Similarly to the FDT LOTOS, the processes of a cTLA system interact via joint synchronous actions. The cTLA patterns are pre-defined and specified as cTLA theorems, where they specify desired protocol behavior by means of safety and liveness properties. cTLA supports the compositional design of protocol procedures based on existing cTLA components. The correctness of these designs can be formally verified based on the theorems defined for each cTLA pattern. The applicability of the approach has been demonstrated for the design of transfer protocols, but it requires a profound experience in cTLA and its semantic rules from the protocol engineer, which is typically not given in practice. Besides, a graphical support for modeling protocols is not given.

Kraemer et al. [16, 17] apply a component based design approach to service design. The developed approach, called *SPACE,* aims at a rapid service engineering using UML collaboration and activity diagrams. The UML collaborations describe the service structure, the activity diagram the service behavior. Based on this, external state machines (ESM) are developed to identify the environment sequences like object and control flow of the system service. This approach is well suited for service engineering,

but it is not quite efficient for protocol design because it does not take such features as entity behaviors, concurrency, and so on into account.

3 On the Modeling of Protocol Design Components

UML as an object-oriented design language promotes the systematic development of systems through 14 types of diagrams [10] which support due to their different and supplementing presentations understandability, extensibility, reusability, and maintainability of the design. In protocol design diagrams like sequence, activity, and state chart diagrams are preferably applied [11, 12]. The *UML sequence diagram* can be used to visualize the message exchange between two systems through sequence charts, while the *UML activity diagram* provides the features to design the internal behavior of the entities. For that reason, we apply both diagrams in a ***protocol design component.*** However, a component is only valid if a concrete abstraction is enclosed on it. This is addressed in this section.

3.1 Protocol Design Components

Communication protocols define the rules how systems in a network communicate with each other. They often apply similar mechanisms and procedures, such as the connection establishment, acknowledged data transfer, flow control, error control, connection release, and others. Such frequently applied mechanisms and procedures can be defined as design components for reuse in various protocol designs. In protocol specifications two approaches are usually applied: the communication- and the behavior- oriented description. The former describes the interaction between the entities, i.e. *the foreseen message exchange*, whereas the latter defines *how this message transfer is accomplished* within the entities. Both views are complementary.

To assemble the communication and behavior-oriented descriptions, we introduce two perspectives: the communication and the behavior perspective, respectively. The *communication perspective* is to act as a design aid, which is often preferred by designers to intuitively design a protocol. It represents a front-end design. The *behavior perspective* precisely defines the respective protocol procedures as an abstract implementation model, i.e. as back-end design. A designer has the possibility to choose the behavior-oriented description for protocol design, but in order to have a standardized framework we realize the assumption made above.

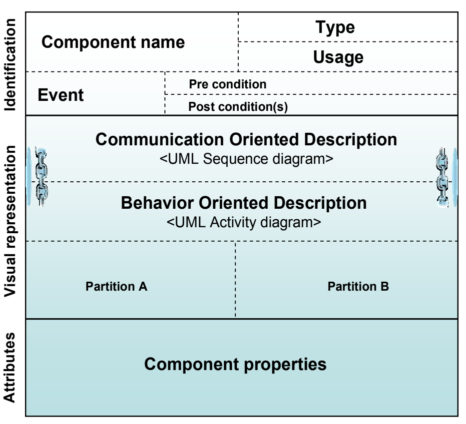


Fig. 1. Template of a protocol component

To provide a concrete abstraction to the component structure we define a template to describe its features. The protocol design component is partitioned into three parts (see Figure 1): identification, visualization, and attributes. The *identification part,* or *tag,*

contains the component name, the type (communication and behavior-oriented description), the use in service or protocol level design, the component triggers (events), and pre-post conditions as parameter constraints. In the *visualization part,* the component behavior is described in UML sequence and activity diagrams. Both presentations are tightly coupled. They describe the same behavior and must be consistent. The partitions A and B indicate the communicating entities. The *attribute part* specifies the expected behavioral properties in linear temporal logics (LTL) the component has to fulfill, e.g. $Data \rightarrow \Diamond Ack$ for an acknowledged data transfer. The attribute part is used to support the verification.

Communication Oriented Description. The presentation is not restricted to point-to-point communication, but it can also include one-to-many or many-to-one relationships. Figure 2 shows an example of UML sequence diagram of a protocol component for a successful connection establishment. The communication-oriented description consists of four lifelines (*initiator, responder, entity A,* and *entity B*). The lifelines represent protocol entities and/or service users. In our example *initiator* and *responder* represent the lifelines of the service users; *entities A* and *B* accordingly the lifelines of the protocol entities. Likewise, other protocol components can be modeled. The service and protocol design components are developed independently and stored in a repository. The semantics to the communication-oriented description is the UML 2.x sequence diagram semantics from [10] in natural English language.

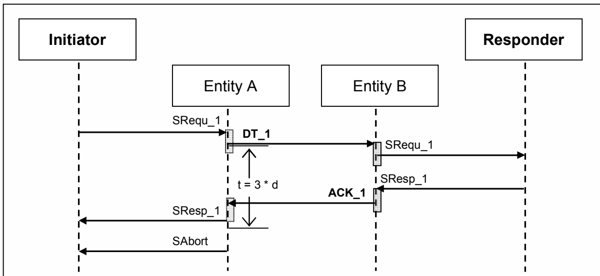
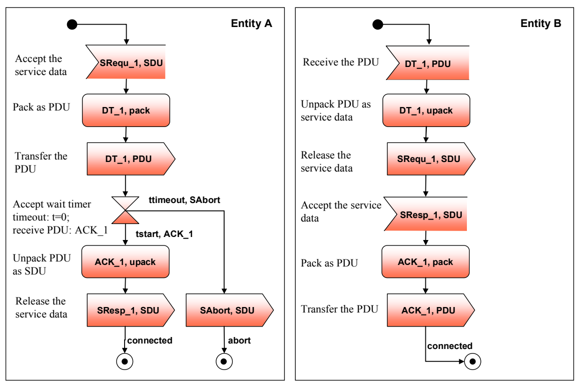


Fig. 2. Communication-oriented description: Successful connection establishment

Behavior Oriented Description. Unlike the communication-oriented description that represents the interaction between entities, the behavior part describes each entity separately. Figure 3.a shows as an example the behavioral description of the *entities A* and *B* through activity diagrams. Figure 3.b gives the identification tag for *entity A*. The identification tag is enclosed within the description as a note. The activities in the behavioral components are composed of nodes like accept event actions, send signal actions, accept wait timers, actions, objects and data flows, decision-makings, forks for concurrent operations, etc.



Id	*B1*
Name	*ConEst_Entity_A*
Type	*Communication*
Usage	*Protocol level*
Event	*ConEst – Entity A*
Pre condi-tion(s)	*connect*
Post condi-tion(s)	*connected/abort*

a) Successful connection establishment b) Identification tag of entity A

Fig. 3. Behavior-oriented description

The components can be developed through any UML 2.x supported tool based on the various protocol functions. Some typical functions that are frequently used in a protocol design are listed in Table 1 as example. Thereafter, the modeled components are stored in a repository, called *protocol component prototypes*, in an XML format for reuse in any standardized UML supported tool.

Table 1. Typical protocol functions and possible reusable components

Protocol functions	Reusable components
Connection establishment	Explicit connection establishment
	Implicit connection establishment
Data Transfer	Stream-based data transfer
	Datagram-based data transfer
Error Control	Go back N
	Selected repeat
…..	…..
Connection release	Explicit connection release
	Abrupt connection release

Since we establish two types of descriptions, there is a need for a *model synchronization* to maintain design consistency between the two descriptions. For brevity, we just explain the logic here. It applies a pre-defined set of *mapping rules*. They provide a syntactical rule to describe the UML sequence and activity diagrams, extracted from the UML document [10]. Thereafter, an algorithm is realized to parse the activity and sequence diagram, and their consistency is evaluated. This corresponds to an UML diagram interchange.

3.2 Formalizing the Semantics of Protocol Design Components

In order to support well-defined protocol compositions as well as automated model transformations into other representations, the semantics of the component has to be defined exactly. This is done by using the *compositional Temporal Logic of Actions*

(cTLA) [8]. cTLA is a formal specification language developed by Herrmann et al. [8] for the specification of transfer protocols. It is based on Lamport's [9] Temporal Logic of Actions. cTLA distinguishes two type of processes: *simple* and *compositional* processes. The *simple cTLA* process is used to model single system resources, while the *compositional cTLA* process is used to model systems and sub-systems as compositions of simple cTLA processes that cooperate by means of synchronously executed process actions. An example is shown in Figure 4. It has a program-like structure with *process* as main function, *constants* and *variables* as declarations, *init* as initial process state, *processes* as sub-functions with an index, and *actions* to define the process behaviors.

The *simple cTLA process* shown in this example describes the logical operation NOT. There are two possible results: *0* or *1*; hence, they are assigned to a constant variable *const_value* under *CONSTANTS.* To determine the current state of the process a transitory variable *var_i* is declared and initialized to *'0'*. Now, a simple cTLA process is triggered through an action *execute*(*var_i: param*). Here, *param* is an input assigned to *var_i*. Based on the given input *var_i*, the result is placed by its complementary value. This determines the process next state, denoted by *var_i'*.

Similarly a *compositional cTLA process* is defined in Figure 4 which comprises the logical operations *NOT*, *AND*, and *OR*. Each operation is independently defined as simple cTLA process and imported in *PROCESSES* by declaring an index pointer *p1: processLogicalNOT*(*param: Value*) (see line 6). It is executed by the action *execute* (*var_1: param*) along with an index pointer *p1* as shown in line 6 of the compositional cTLA process definition. The control flow transfers from the compositional to simple cTLA process. This corresponds to a subroutine call in programming languages.

Simple cTLA Process	Compositional cTLA Process

1.**PROCESS** processLogicalNOT
(*param*: value)
2.**CONSTANTS** const_value = {"0","1"};
3.**VARIABLES** var_i;
4.**INIT** ≜ var_i = 0;
5.**ACTIONS**
execute(var_i: *param*) ≜
var_i = 0 ∧ var_i ∈ const_value
⇒ var_i'=1;
var_i = 1 ∧ var_i ∈ const_value
⇒ var_i'=0;
6.**END**
	1.**PROCESS** processLogicalOperation
	(*param*: Values, *paramType:*ValueType)
	2.**IMPORT** moduleName(param)
	3.**CONSTANTS** const_value = {"0","1"};
	4.**VARIABLES** var_i, var_j;
	5.**INIT**≜ var_i = 0;
	6.**PROCESSES**
	p1: processLogicalNOT(*param*: Value);
	p2: processLogicalAND(*param*: Value1, Value2);
	p3: processLogicalOR(*param*: Value1, Value2);
	7.**ACTIONS**
	execute(*param, paramType*) ≜
	paramType ∈ {"NOT", "AND", "OR"}:
	paramType = "NOT" ⇒ p1.execute(var_i: *param*);
	paramType = "AND" ⇒ p2.execute(var_i, var_j: *param*);
	paramType = "OR" ⇒ p3.execute(var_i, var_j: *param*);
	8.**END**

Fig. 4. A simple and a compositional cTLA process

The reasons for using cTLA to formalizing the semantics of UML activity diagrams are the following. (i) The formal semantics enables an exact interpretation of the system design specification. (ii) As a temporal event based system, it is possible to

rewrite cTLA processes in a canonical form. This form can be used to verify the system behaviors by introducing an appropriate verification mechanism, e.g. model checking. It is also possible to formulate time-ordering events as properties to prove whether it holds in the cTLA process or not. (iii) Due to its standard structure, it is appropriate for model transformations into other formal description techniques, such as Lotos, and into verification languages like PROMELA. The approaches from [8,18] also addressed the importance of using cTLA for protocol verification. By considering the above advantages, we favor cTLA as semantics definition for UML activity diagrams. The formalization of the semantics is confined to the activity diagrams in the following. The communication-oriented description serves as an interface description to promote an intuitive protocol design. With the selection and adaptation of the components, the behavior-oriented description is automatically generated. It forms the basis for the further protocol development.

3.3 Operational and Functional Semantics

Two different types of semantics are applied in our approach: an *operational* and a *functional* semantics. An operational semantics is a way to give meaning to a design /program language in a rigorous way. It describes how a language can be interpreted. In contrast to the operational semantics, the functional semantics is based on syntactic transformations of the design/program and simple operations on discrete data [20]. Likewise, we apply the operational semantics to give a valid meaning for simple operations of the UML activity nodes that are specified in the UML document [10]. The functional semantics is used to unify the simple operations and to obtain a formalized specification. Defining such kind of semantics has the following benefits. Now, the designer is free to model the system in any standardized UML supported tool. Thereafter, the designer can employ functional semantics to generate a formal specification of his/her specification.

3.3.1 Simple cTLA Processes as Operational Semantics

Simple cTLA processes are used to define single system resources. Likewise, we apply simple cTLA processes to define the operational semantics of the activity nodes assigning to each activity node a pre-defined simple cTLA process. The definitions can be re-used in different system designs just by updating the activity data. We formalize a subset of UML activity nodes, such as *init, final, fork, join, merge, decision, send signal, accept wait, accept event, control edges,* and *action* nodes, which are often used in protocol design. According to the UML document [10], the activities have a Petri net like semantics, i.e. the semantics is based on a token flow. Hence, an *activity* describes a *state chart* with the *token movements* as *transitions,* and the *placement of tokens* in the graph as *states*. In cTLA, the token movement and tokens can be represented by means of process parameters. The activity token (AT) is assumed as a set of triggering tokens for the activities. These tokens represent the data flow among activities. Due to lack of space, we show the semantics definition only for the *accept event action* here.

The **accept event action** is a receiving node from an activity diagram/partition (parttion refers to the participants, e.g. *entity* B). The data received from another activity partition is modeled through the action trigger *accept*. A variable *rec* (receive)

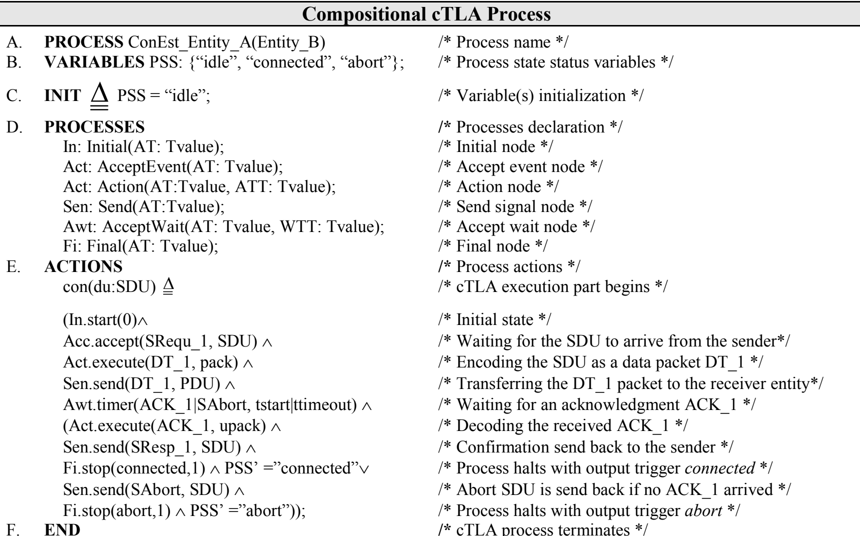
is used to store the data for the action trigger. It is an auxiliary token for the entire activity. The parameter *rd* is the receivable data and mapped to the token *rec'*, as a next state.

Activity node: Accept event action

```
PROCESS AcceptEvent(AT: Any)
VARIABLES rec;
INIT ≜  rec = NULL;
ACTIONS
   accept(rd: AT) ≜ rec'=rd ∧ rec ≠NULL;
END
```

3.3.2 Compositional cTLA Process as Functional Semantics

The objective of the functional semantics is to formalize the entire design specified in an activity diagram based on the simple cTLA processes defined for the independent activity nodes. At this point, the behavior of an activity diagram is unknown, i.e. the activity data and the execution sequence are unidentified. By lacking this feature, it is not possible to check whether the design specifications are semantically valid or not. For this, an automated mechanism, called *Activity to cTLA* (*A2cTLA*) *process generator* [19], was developed as a tool to map the activity diagrams onto cTLA. There may exist various ways to construct a compositional cTLA process, but we intend to use the semantics definition as a base during the model transformations. Hence, we realize the compositional cTLA process generation similarly to a compiling process. Given an activity diagram in an XML format and the simple cTLA processes of the activity nodes, the tool generates its equivalent compositional cTLA process. . Due to the space limitation, technical details of the transformation cannot be given here. They are described in detail in [19]. We confine here to an example (see Figure 5) the compositional cTLA that corresponds to the activity diagram of *entity A* in Figure 3.a.



Compositional cTLA Process

```
A.  PROCESS ConEst_Entity_A(Entity_B)             /* Process name */
B.  VARIABLES PSS: {"idle", "connected", "abort"};  /* Process state status variables */
C.  INIT ≜ PSS = "idle";                            /* Variable(s) initialization */
D.  PROCESSES                                       /* Processes declaration */
      In: Initial(AT: Tvalue);                      /* Initial node */
      Act: AcceptEvent(AT: Tvalue);                 /* Accept event node */
      Act: Action(AT:Tvalue, ATT: Tvalue);          /* Action node */
      Sen: Send(AT:Tvalue);                         /* Send signal node */
      Awt: AcceptWait(AT: Tvalue, WTT: Tvalue);     /* Accept wait node */
      Fi: Final(AT: Tvalue);                        /* Final node */
E.  ACTIONS                                         /* Process actions */
      con(du:SDU) ≜                                 /* cTLA execution part begins */

      (In.start(0)∧                                 /* Initial state */
      Acc.accept(SRequ_1, SDU) ∧                    /* Waiting for the SDU to arrive from the sender*/
      Act.execute(DT_1, pack) ∧                     /* Encoding the SDU as a data packet DT_1 */
      Sen.send(DT_1, PDU) ∧                         /* Transferring the DT_1 packet to the receiver entity*/
      Awt.timer(ACK_1|SAbort, tstart|ttimeout) ∧    /* Waiting for an acknowledgment ACK_1 */
      (Act.execute(ACK_1, upack) ∧                  /* Decoding the received ACK_1 */
      Sen.send(SResp_1, SDU) ∧                      /* Confirmation send back to the sender */
      Fi.stop(connected,1) ∧ PSS' ="connected"∨     /* Process halts with output trigger connected */
      Sen.send(SAbort, SDU) ∧                       /* Abort SDU is send back if no ACK_1 arrived */
      Fi.stop(abort,1) ∧ PSS' ="abort"));           /* Process halts with output trigger abort */
F.  END                                             /* cTLA process terminates */
```

Fig. 5. cTLA process for the connection establishment component

4 Modeling a Protocol Design Specification

In order to assemble the components for a protocol design specification we use a *presentation interface* that is based on UML interaction overview (IO) diagrams [10]. It is a combination of the sequence and activity diagrams. It has the feature to design multiple sequence diagrams in a single window which can be linked together through the activity nodes. It allows the protocol designer to use both the communication and the behavior perspective in one presentation when assembling the components.

4.1 Communication Perspective

The presentation interface consists of two functions namely: (i) *extract and build* - various service or protocol components are extracted from the component repository and reassembled through the control lines, (ii) *refinement* – for adapting.

(i) Extract and build. Before illustrating the principle of this step, we describe certain constraints that should be considered while assembling the protocol components. The constraints are used to model and interpret the protocol design in a standard way.

1. An initial node should be placed at the beginning of protocol design to ensure that the design has a starting point.
2. A final node should be placed at the end of the protocol design to ensure that the design has an end point. This does not apply to cyclic protocols.
3. Activity node constraints [10] that are used to design an activity diagram should be followed.

First, the components are extracted from the *protocol component prototypes* repository and placed on the presentation interface, i.e., with a group of sequence diagrams. An initial node is placed in the beginning of the presentation interface. Later the designer reorders the components accordingly to the protocol requirements (see Figure 6). The communication flow among the extracted components is modeled using the activity connection nodes like object flows, and control flows. The protocol features can be further modeled with the help of forks for concurrent operations, joins for concatenating concurrent operations, decisions for conditions, merge for combining multiple control flows. For design information, several comment nodes can be introduced to indicate the usage of component adaptations. On a component assembly, at last a final node is placed and linked with the component's control flow to indicate the end of protocol specification.

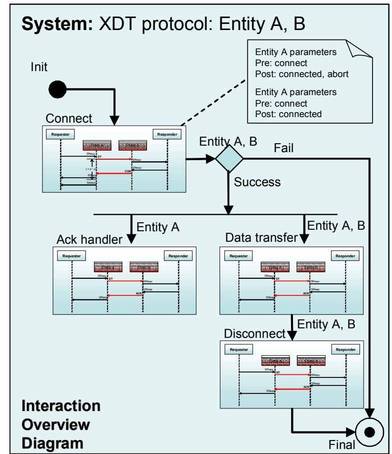


Fig. 6. Communication Perspective

(ii) Refinement. The service or protocol design components or specification may need some differentiation regarding their default names. Each designer uses own names and notations to describe the protocol or service primitives. There exists one rule to accomplish this task. The notions can be changed, but not the values. For instance, the component in Figure 3.a has a notion *SRequ_1*. This can be changed into *ServiceRequest_1*, *ServiceIndication_1* etc. The value followed by the underscore is considered as a sequence number. This is to represent a standard notion for the service and protocol primitives. Apart from the naming convention, there might be a situation, where a component does not fit in the compositions. For example, a protocol specification may require additional features in the data transfer component, e.g. a pause. This feature may not be available in the extracted component. For this, the designer has an option to add this feature to the data transfer component.

4.2 Behavior Perspective

Whenever a communication description of a design component is extracted from the repository its corresponding behavior-oriented description is extracted and placed in the *behavior perspective* as basis for the further protocol development. For example, the behavior descriptions of *entities* A and B in Figure 7 are generated with the communication-oriented description of Figure 6. This generation is carried out simultaneously during the protocol design.

There is another problem which has to be taken into account during design. Protocols are divided in symmetric or asymmetric protocols regarding the behavior of the protocol entities. In symmetric protocols the entities show the same behavior, while in asymmetric ones it is different. In a symmetric protocol design, the designer can therefore use the same control flow in both entities. Thus the behavior description can be automatically derived. In an asymmetric protocol design this cannot be done. For instance, consider the asymmetric protocol design specification shown in Figure 6. There exist four components: *connect*, *ack handler*, *data transfer*, and *disconnect*. Here, the *ack handler* component should only exist in *entity A*. If a designer uses the same control flow for the communication and behavior perspective adaptations, one could obtain a wrong protocol design for *entity* B because the *ack handler* component does not exist here. To overcome this situation we define a textual constraint near to the control flow during the communication description adaptations. The constraints are visible to the designer

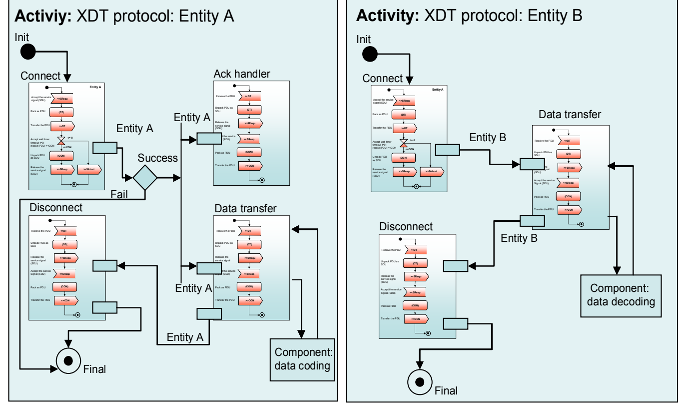


Fig. 7. Behavior Perspective

through an identification tag as shown in Figure 6 and Figure 3.b. Based on this constraint, the components are differentiated and adapted in the behavior perspective (see Figure 7), i.e. whether it belongs to *entity* A or B. To sum up the presentation interface has a support of UML *sequence*, *activity*, and *interaction overview diagrams*.

4.3 Evaluation Example

To illustrate the workflow of our method, we use an example case study of the *eXample Data Transfer* (XDT) protocol [13] which is being used as teaching protocol. XDT works on a distributed environment to transfer large files over an unreliable media using the go back N principle (retransmission of messages, when some messages are not successfully transferred). The protocol description consists of a service and a protocol specification, which both include a data format specification. The XDT protocol functionalities are shown in Figure 8. The sender makes an initiative for data transmission to the receiver by means of an XDATrequ service primitive. The new connection is indicated by an XDATind primitive. The protocol indicates the successful connection set up to the sender by XDATconf. After this, the data are transferred by means of a DT message. However in certain cases, the service provider may not preserve the order of the data units. In this case, the ABO message is initialized to abort the connection. This is indicated to the users by a XABORTind primitive. XBREAKind is initialized to stop the acceptance of data flow from the user for a certain period, if the go back N message buffer is full.

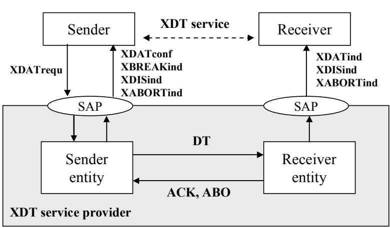


Fig. 8. XDT service and protocol

The end of transmission is indicated by setting the parameter *eom* in the final data unit of XDATrequ and XDATind primitives. The connection is released implicitly, indicated by an XDISind primitive at both sides after successfully transmitting the last data unit. The further explanation of the XDT protocol can be found in [13].

In principle, the service and protocol components are stored together in the repository. Whenever the designer models the protocol design components, its corresponding services are also specified concurrently. However, it is also possible that the designer can model the service and protocol separately. In the following, we demonstrate the service and protocol design in parallel. According to the XDT protocol requirements; there is a need for components for *connection set up*, *message transfer*, *acknowledgement handler*, *connection abort*, *go_back_N*, and *connection release*. At first, we look for appropriate components in the repository to fulfill the above requirements.

Communication Perspective. We assume that the following components *connect* (C1), *transfer* (C2), *acknowledgment confirmation* (C3), *abort_entity A* (C4), *abort_ entity B* (C5), *go back N* (C6), and *disconnect* (C7) exist in the protocol component repository and may fit to the XDT protocol design. Hence, these seven components are extracted and placed on the communication perspective (see C1 to C7 in Figure 9).

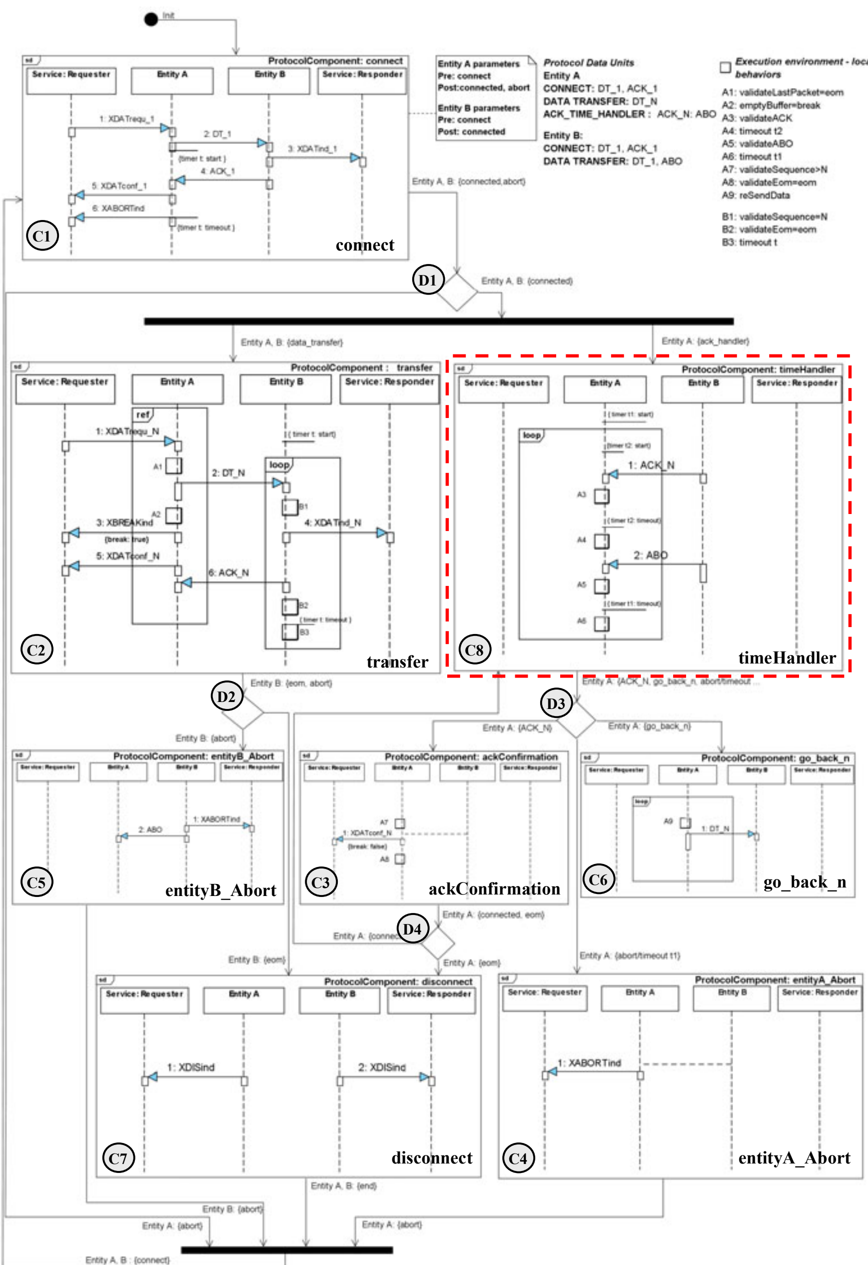


Fig. 9. XDT protocol specification: Communication-oriented description

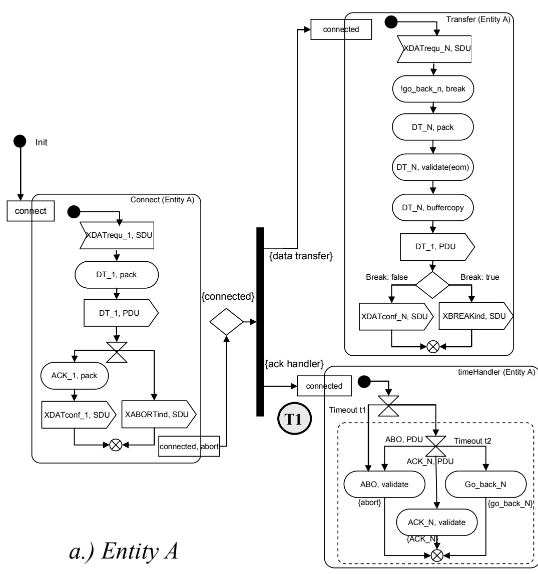
An initial node is placed at the top of the extracted components to indicate the start state of the XDT protocol design specification. Next, we assemble the components through the control flows. A control flow is used to link the initial node and the connect component. The connect component consists of two different outputs: *connected* and *abort* (see identification tag in Figure 9). We introduce therefore a decision node (D1) to validate them. The connect component is linked with the decision node (D1) using another control flow. There exist two parallel parts in the XDT protocol: the *data transfer* and the *acknowledge handler*. The data transfer part is responsible to send and receive messages as well as to validate whether the sent data is the end of message or not. The acknowledgement handler is used to check the message sequence, to trigger message retransfer, connection abort, and to handle time-out events. To specify the two parallel parts we introduce a fork node and link it with the decision node (D1).

For brevity, we consider the acknowledgment handler and its adaptations. Adapting the four events, such as sequence validation, message retransfer, connection abort, and time-out, from the fork node may lead a design overload. The reason is the following. The four events execute repeatedly based on a global timer and receivable messages like ACK (acknowledgement), and ABO (abort) from *entity B*. For example, if an acknowledgment is not received for a certain time the handler assumes that the message is not successfully transferred. In this case, the go_back_N mechanism is activated and the messages from N^{th} one are retransmitted. If an abort signal is received from entity *B* then an abort is triggered. Likewise, the time-out also leads to an abort. Hence, instead of specifying four events in four loops they can be capsulated into a single activity. For this, we model a new component (see dashed rectangle in Figure 9), called the *timeHandler,* using two timers t1 (to receive data), t2 (for go_back_n) and a loop activity as shown in Figure 9 (see C8). Now, the *timeHandler* component possesses the following output parameters, i.e. ACK_N, go_back_N, ABO, and time-out. In most cases, ABO and time-out in the XDT protocol result in an exception condition. Hence, we consider the ABO and timeout to be a single output parameter.

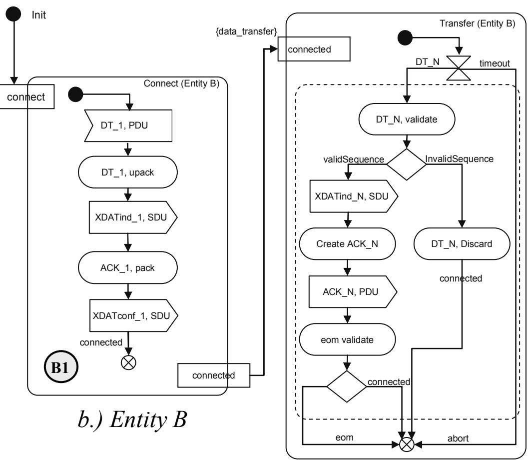
Next, we continue the component adaptations. To validate the *timeHandler* parameters, another decision node (D3) is placed near to the *timeHandler* component and linked with a control flow. In the following, we explain the ACK_N parameter adaptation.

ACK_N, if an acknowledgment is received from *entity B* then a control flow is used to link the decision node (D3) and ackConfirmation (C3) component. Next, a decision node (D4) is introduced to validate the outputs, such as *connected* and *eom* from the ackConfirmation component. If it is connected a control flow is used to link the decision node (D4) and the *timeHandler* (C8) component. This is to specify that the data transfer between the two entities will be accomplished. In the other case, i.e. *eom*, a control flow is used to link the decision node (D4) and the disconnect component to specify the connection release. Likewise, other parameters are evaluated and linked appropriately. Finally, a control flow from the join node and to the connect (C1) component is linked to provide the next service in the XDT protocol.

Behavior Perspective. Now we consider the behavior of the communication entities, i.e. the specification has to be partitioned into two parts, for *entities A* and *B*. For instance, Figure 10.a depicts the behavioral design for entity *A* that corresponds to Figure 9. For brevity, we show only some few components in Figure 10. The control flows for the partitioned components are independently visualized here. For instance, consider the *timeHandler* component in Figure 9 that should exist only in *entity A*. The control flow for the *timeHandler* (T1) component of entity *A* in the behavior perspective is straightforward, i.e. the designer can use the same control flow.



a.) Entity A



b.) Entity B

Fig. 10. XDT protocol specification: Behavior-oriented description

Since, the XDT protocol is asymmetric, the *timeHandler* component in *entity B* does not exist, i.e., there is no need of a fork node. The corresponding control flows are discarded automatically. Besides, there exist only one output parameter from the connect component (see B1 in Figure 10.b) in *entity B*, and hence the decision node is discarded. These two facts are clearly visible in Figure 10.b. Likewise, the other components are linked together with respect to the constraints specified in the communication -oriented design. For the XDT protocol design, seven components are reused from the repository, while one component is newly introduced, i.e., *timeHandler* (see Figure 9). This considerably shortens the development time. As result we get a communication and a behavior-oriented XDT specification. Besides, the refactoring of the protocol design is simple too. If the designer wants to change a given specification, e.g. using *selected repeat* instead of *go_back_N*, he/she has simply to

replace the component *C6* by the *selected repeat* component from the repository and use the existing control flows to link the components.

5 Final Remarks

We have presented a component-oriented approach for a UML based design and development of communication protocols. The use of UML in the protocol development has several advantages compared to the traditional FDT based approaches. Unlike FDTs which enforce a semantic-oriented description UML better supports an intuitive modeling of the protocols and the related services which allows in particular a reuse of the designs. In contrast to FDTs however, UML does not possess a formal semantics which is needed for a unique interpretation of the specifications and the transformation in other representations. For that reason, we first introduced a formal semantics for activity diagrams using the compositional Temporal Logic of Actions (cTLA). The formalization is done in two steps. First we introduce an operational semantics for the activity diagram nodes using simple cTLA processes. The functional semantics of the specification is derived by mapping the activity diagrams into a compositional cTLA process. Based on this formal definition we introduced a component based design approach. It applies two perspectives: a communication perspective using sequence diagrams and a behavior perspective using activity diagrams. Both views are complementary and synchronized with each other. The communication perspective supports the intuitive design by representing the interactions between the protocol entities. The behavior perspective describes how these interactions are "implemented" in the entities. The activity diagram specifications form the basis for the further development steps. We have demonstrated with an example how the components can be used to set up a protocol design specification and how adaptations to the context have to be taken into account. Moreover, we use the UML 2.x compatible format, i.e. XML, which helps the design to be interoperable between multiple UML tool vendors.

The protocol design specification is dependable only if they are proved for correctness properties, such as deadlock freedom, livelock freedom, etc. Illustrating an approach to design verification goes out of the scope to this paper. Here we only outline the principle. Each component possesses a concrete abstraction and should be verified for design errors by applying an appropriate verification mechanism. At the same time, the component adaptations should be verified to check whether the components hold desired behavior or not. For this, we favor incremental verification that helps to detect the design errors concurrently, by tracking the component adaptations. The inclusion of the property specifications in the attribute part assists this verification. The component is transformed into PROMELA for formally proving the correctness of the properties with the help of the model checker SPIN [14]. Currently, we are developing the protocol design components and implementing the approach using a visual paradigm suite [15]. Next, it is planned to implement the design and verification process under single window system using the eclipse environment.

References

1. ITU-T Recommendation Z.100.: Specification and Description Language SDL (2002)
2. ISO: Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour, IS 8807 (2006)
3. Jaragh, M., Saleh, K.A.: Protocols Modeling Using the Unified Modeling Language. In: Proceedings of IEEE Region 10 International Conference, Singapore (2001)
4. Pärssinen, J.: Turunen.M.: Patterns for Protocol System Architecture. In: Proceedings of the 7th Conference on Pattern Languages of Programs, Illinois, USA (2000)
5. Gotzhein, R.: Consolidating and Applying the SDL-pattern approach: A Detailed Case Study. In: Information and Software Technology, vol. 45 -11. Elsevier Sciences, Amsterdam (2003)
6. Geppert, B., Rößler, F.: The SDL pattern approach – A Reuse-Driven SDL Design Methodology. Computer Networks 35(6), 627–645 (2001)
7. Byun, Y., Sanders, B.A.: A Pattern Based Development Methodology for Communication Protocols. In: Proc. of the ACM Symposium on Applied Computing, New York (2005)
8. Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks 34 (2000)
9. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software. Pearson Education, Inc., London (2002)
10. Object Management Group.: UML Superstructure - Specification Standard Document. OMG Unified Modeling Language (OMG UML) (February 2009)
11. Thramboulidis, K., Mikroyannidis, A.: Using UML for the Design of Communication Protocols: The TCP case study. In: IEEE International Conference on Software, Telecommunications and Computer Networks, Dubrovnic, Croatia (2003)
12. Patel, D.: Object-Oriented Design of an Embedded Communication Protocol in UML. A Technical Report, Design of Embedded Systems, University of California, Berkeley (1999)
13. eXample Data Transfer Protocol, `http://www.protocol-engineering.tu-cottbus.de/`
14. Holzmann, G.J.: The SPIN Model Checker. Addison-Wesley, Reading (2006)
15. Visual Paradigm UML Suite, `http://www.visual-paradigm.com/product/vpuml/`
16. Kraemer, F.A., Herrmann, P.: Service Specification by Composition of Collaborations - An Example. In: Proceedings of the WI-IAT Workshops, Hong Kong, P.R. China (2006)
17. Kraemer, F.A., Slåtten, V., Herrmann, P.: Model-Driven Construction of Embedded Applications based on Reusable Building blocks. In: Reed, R., Bilgic, A., Gotzhein, R. (eds.) SDL 2009: Design for Motes and Mobiles. LNCS, vol. 5719, pp. 1–18. Springer, Heidelberg (2009)
18. Graw, G., Herrmann, P., Krumm, H.: Verification of UML-Based Real-Time System Designs by Means of cTLA. In: Proceedings of the Third IEEE international Symposium on Object-Oriented Real-Time Distributed Computing. IEEE Computer Society, Los Alamitos (2000)
19. Kaliappan, P.S., König, H., Schmerl, S.: Formal Methods Integration to UML-based Design Specification. Submitted to SLE 2010, Eindhoven, The Netherlands (2010)
20. Plotkin, G.D.: A Structural Approach to Operational Semantic. Journal of Logic and Programming in Structural Operational Semantics 60-61 (December 2004)

Laws of Pattern Composition

Hong Zhu and Ian Bayley

Oxford Brookes University, Wheatley Campus,
Wheatley, Oxfordshire OX33 1HX, UK
hzhu@brookes.ac.uk, ibayley@brookes.ac.uk

Abstract. Design patterns are rarely used on their own. They are almost always to be found composed with each other in real applications. So it is crucial that we can reason about their compositions. In our previous work, we defined a set of operators on patterns so that pattern compositions can be represented as expressions on patterns. In this paper, we investigate the algebraic properties of these operators, prove a set of algebraic laws that they obey, and use the laws to show the equivalence of pattern compositions.

Keywords: Design patterns, Pattern composition, Formal methods, Algebraic laws, First order logic.

1 Introduction

Design patterns are codified reusable solutions to recurring design problems [9,1]. Many such patterns have been identified, documented, catalogued [6] and included in software tools [11,16,14]. Although each is specified separately, they are usually to be found composed with each other with overlaps except in trivial cases [17]. However, while the importance of pattern compositions has been widely recognised, it has not been studied intensively. This is perhaps partly because the patterns have been documented informally.

In the past few years, significant progress has been made by several researchers in the formalisation of design patterns. Several approaches have been advanced in the literature [15,13,19,7,10,5]. In spite of the differences in the formalisms used by these approaches, the basic ideas underlying them are similar. In particular, a specification of a pattern usually consists of statements on the common structural features and, sometimes, behavioural features of its instances. The structural features of a pattern are typically specified by assertions on the existence of certain types of components in the pattern. The configuration of the elements is also described, in terms of the static relationship between them. The behavioural features are normally defined by assertions on the temporal orders of the messages exchanged between the components as manifested in the designs of systems. This formalisation lays a foundation for systematically and formally investigating the composition of design patterns.

However, very few authors have investigated composition formally. In [18], Taibi illustrated the concept of pattern composition in his framework of pattern formalisation with an example. In [3], we formally defined a universal pattern composition operator. In [22], we extended and revised the work, but took a radically different approach.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 630–645, 2010.
© Springer-Verlag Berlin Heidelberg 2010

We replaced the single operator with a set of simpler operators that express composition when used together. A case study was also reported there to demonstrate the expressiveness of the operators. In this paper, we continue the work in this direction by investigating how to reason about pattern compositions, such as how to determine whether two pattern compositions are equivalent. We will prove a set of algebraic laws that these operators obey and demonstrate, with an example, how to prove equivalence of pattern compositions by equational reasoning.

The particular formalism that we will use in this paper to define operators and to prove their algebraic laws is that advanced in our previous work. This uses the first-order logic induced from the abstract syntax of UML defined in GEBNF [20,21] to define both the structural and behavioural features of design patterns. In this way, we have already formally specified the 23 patterns in the classic Gang of Four (hereafter referred to as GoF) book [9], and we have specified variants too [2,4,5]. We have also constructed a prototype software tool to check whether a design represented in UML conforms to a pattern [23,24]. It is worth noting that the definitions of the operations and the algebraic laws proved in this paper are independent of the formalism and thus can equally well be applied to others such as OCL [8], temporal logic [18], and so on, but the results may be less readable. In particular OCL would need to be applied at the meta-level to assert the existence of the required classes and methods.

The remainder of the paper is organised as follows. Section 2 reviews our approach to formalisation and lays the theoretical foundation for our proofs. Section 3 outlines the set of operations on design patterns. Section 4 presents the algebraic laws that they obey. Section 5 outlines the use of laws in equational reasoning about the equivalence of pattern compositions with an example. Section 6 concludes the paper with a discussion of related works and future work. For the sake of readability and space, the proofs of the algebraic laws are removed from the body of the paper and some are given in the appendix.

2 Background

This section briefly reviews our approach to the formal specification of design patterns. It is based on meta-modelling in the sense that each pattern is a subset of the design models having certain structural and behavioral features. Readers are referred to [2,4,23,5] for details.

2.1 Meta-modelling in GEBNF

Our approach starts by defining the domain of all models with an abstract syntax written in the meta-notation Graphic Extension of BNF (GEBNF) [20]. GEBNF extends the traditional BNF notation with a 'reference' facility to define the graphical structure of diagrams. In addition, each syntactic element in the definition of a language construct is assigned an identifier (called a *field name*) so that a first-order language (FOL) can be induced from the abstract syntax definition [21].

For example, the following are some example syntax rules in GEBNF for the UML modelling language.

$$\begin{array}{ll} ClassDiag & ::= classes : Class^{+}, assocs, inherits, compag : Rel^{*} \\ Class & ::= name : String, [attrs : Property^{*}], [opers : Operation^{*}] \\ Rel & ::= [name : String], source : End, end : End \\ End & ::= node : \underline{Class}, [name : String], [mult : Multiplicity] \end{array}$$

The first line defines a class diagram as consisting of a non-empty set of classes and a collection of three relations on the set. Here $classes$, $assocs$, $inherits$ and $compag$ are field names. Each field name is a function. For example, $classes$ is a function from a $ClassDiag$ to the set of class nodes in the model. Functions $assocs$, $inherits$ and $compag$ are mappings from a class diagram to the sets of association, inheritance and composite/aggregate relations in the model. The non-terminal $Class$ in the definition of End is a reference occurrence. This means that the node at the end of a relation must be an existing class node in the diagram, not a newly introduced class node. The definitions of the class diagrams and sequence diagrams of UML in GEBNF can be found in [5]. Table 1 gives the functions used in this paper that are induced from these definitions as well as those that are based on them. A formal more detailed treatment of this can be found in [5].

Table 1. Some Functions Induced from GEBNF Syntax Definition of UML

ID	Domain	Function
Functions directly induced from GEBNF syntax definition of UML		
$classes$	Class diagram	The set of class nodes in the class diagram
$assocs$	Class diagram	The set of association relations in the class diagram
$inherits$	Class diagram	The set of inheritance relations in the class diagram
$compag$	Class diagram	The set of composite and aggregate relations in the class diagram
$name$	Class node	The name of the class
$attr$	Class node	The attributes contained in the class node
$opers$	Class node	The operations contained in the class node
sig	Message	The signature of the message
Functions defined based on induced functions		
$X \longrightarrow\!\!\triangleright^{+} Y$	Class	Class X inherits class Y directly or indirectly
$X \longrightarrow^{+} Y$	Class	There is an association from class X to class Y directly or indirectly
$X \diamond\!\!\longrightarrow^{+} Y$	Class	There is an composite or aggregate relation from X to Y directly or indirectly
$isInterface(X)$	Class	Class X is an interface
$CDR(X)$	Class	No messages are send to a subclass of X from outside directly
$subs(X)$	Class	The set of class nodes that are subclasses of X
$calls(x, y)$	Operation	Operation x calls operation y
$isAbstract(op)$	Operation	Operation op is abstract
$fromClass(m)$	Message	The class of the object that message m is sent from
$toClass(m)$	Message	The class of the object that message m is sent to
$X \approx Y$	Operation	Operations X and Y share the same name

2.2 Formal Specification of Patterns

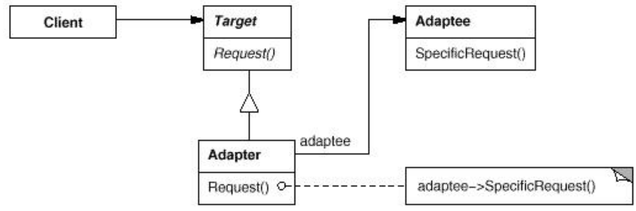
Given a formal definition of the domain of models, we can for each pattern, define a predicate in first-order logic to constrain the models such that each model that satisfies the predicates is an instance of the pattern.

Definition 1. *(Formal specification of DPs)*
A formal specification of a design pattern is a triple $P = \langle V, Pr_s, Pr_d \rangle$, where Pr_s and Pr_d are predicates on the domain of UML static class diagrams and dynamic sequence diagrams, respectively, and V is a set of declarations of the variables that are free in the predicates Pr_s and Pr_d. Let $V = \{v_1 : T_1, \cdots, v_n : T_n\}$. The semantics of the specification is the closed formula in the following form.

$$\exists v_1 : T_1 \cdots \exists v_n : T_n \cdot (Pr_s \wedge Pr_d) \tag{1}$$

In the sequel, we write $Spec(P)$ to denote the predicate (1) above, $Vars(P)$ for the set of variables declared in V, and $Pred(P)$ for the predicate $Pr_s \wedge Pr_d$.

For example, Fig. 1 shows the specification of the Object Adapter design pattern. The class diagram from the GoF book has been included for the sake of readability.



Specification 1 *(Object Adapter Pattern)*
Components

1. $Target, Adapter, Adaptee \in classes$,
2. $requests \subseteq Target.opers$,
3. $specreqs \subseteq Adaptee.opers$

Static Conditions

1. $Adapter \longrightarrow\!\!\!\triangleright^+ Target$, $Adapter \longrightarrow^+ Adaptee$,
2. $CDR(Target)$

Dynamic Conditions

1. $\forall o \in requests \cdot \exists o' \in specreqs \cdot (calls(o, o'))$

Fig. 1. Specification of Object Adapter Pattern

Fig. 2 gives the specification of the $Composite$ pattern. Both patterns will be used throughout the paper.

2.3 Reasoning about Patterns

We often want to show that a concrete design really conforms to a design pattern. This is a far from trivial task for some other formalisation approaches. For us though, the

Specification 2 *(Composite)*
Components

1. $Component, Composite \in classes,$
2. $Leaves \subseteq classes,$
3. $ops \subseteq Component.opers$

Static Conditions

1. $ops \neq \emptyset$
2. $\forall o \in ops.isAbstract(o),$
3. $\forall l \in Leaves \cdot (l \longrightarrow\!\!\triangleright^{+} Component \wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
4. $isInterface(Component)$
5. $Composite \longrightarrow\!\!\triangleright^{*} Component$
6. $Composite \diamond\!\!\longrightarrow^{+} Component$
7. $CDR(Component)$

Dynamic Conditions

1. *any call to* $Composite$ *causes follow-up calls*

$$\forall m \in messages \cdot \exists o \in ops \cdot (toClass(m) = Composite \wedge m.sig \approx o \Rightarrow \exists m' \in messages \,.\, calls(m, m') \wedge m'.sig \approx m.sig)$$

2. *any call to a leaf does not*

$$\forall m \in messages \cdot \exists o \in ops \cdot toClass(m) \in Leaves \wedge m.sig \approx o \Rightarrow \neg\exists m' \in messages \,.\, calls(m, m') \wedge m'.sig \approx m.sig)$$

Fig. 2. Specification of the Composite Pattern

use of predicate logic makes it easy and we formally define the conformance relation as follows.

Let m be a model and pr be a predicate. We write $m \models pr$ to denote that predicate pr is true in model m. Readers are referred to [21] for the formal definition of $m \models pr$.

Definition 2. *(Conformance of a design to a pattern)*
Let m be a model and $P =< V, Pr_s, Pr_d >$ be a formal specification of a design pattern. The model m conforms *to the design pattern as specified by P if and only if $m \models Spec(P)$.* □

To prove such a conformance we just need to give an assignment α of variables in V to elements in m and evaluate $Pred(P)$ in the context of α. If the result is $true$, then the model satisfies the specification. This is formalised in the following lemma.

Lemma 1. *(Validity of conformance proofs)*
A model m conforms to a design pattern specified by predicate P if and only if there is an assignment α from $Vars(P)$ to the elements in m such that $Eva_\alpha(m, Pred(P)) = true$. □

A software tool has been developed that employs the first order logic theorem prover *SPASS*. With it, proofs of conformance can be performed automatically [23,24].

Given a formal specification of a pattern P, we can infer the properties of any system that conforms to it. Using the inference rules of first-order logic, we can deduce that $Spec(P) \Rightarrow q$ where q is a formula denoting a property of the model. Intuitively, we expect that all models that conform to the specification should have this property and the following lemma formalises this intuition.

Lemma 2. *(Validity of property proofs)*
Let P be a formal specification of a design pattern. $\vdash Spec(P) \Rightarrow q$ implies that for all models m such that $m \models Spec(P)$ we have that $m \models q$. □

In other words, every logical consequence of a formal specification is a property of all the models that conform to the pattern specified.

There are several different kinds of relationships between patterns. Many of them can be defined as logical relations and proved in first-order logic. Specialisation and equivalence are examples of them.

Definition 3. *(Specialisation relation between patterns)*
Let P and Q be design patterns. Pattern P is a specialisation *of Q, written $P \preccurlyeq Q$, if for all models m, whenever m conforms to P, then, m also conforms to Q.* □

Definition 4. *(Equivalence relation between patterns)*
Let P and Q be design patterns. Pattern P is equivalent *to Q, written $P = Q$, if $P \preccurlyeq Q$ and $Q \preccurlyeq P$.* □

By Lemma 1, we can use inference in first-order logic to show specialisation.

Lemma 3. *(Validity of proofs of specialisation relation)*
Let P and Q be two design patterns. Then, we have that

1. *$P \preccurlyeq Q$, if $Spec(P) \Rightarrow Spec(Q)$, and*
2. *$P = Q$, if $Spec(P) \Leftrightarrow Spec(Q)$.* □

Furthermore, by Definition 1 and Lemma 3, we can prove specialisation and equivalence relations between patterns by inference on the predicate parts alone if their variable sets are equal.

Lemma 4. *(Validity of proofs of predicate relation)*
Let P and Q be two design patterns with $Vars(P) = Vars(Q)$. Then $P \preccurlyeq Q$ if $Pred(P) \Rightarrow Pred(Q)$, and $P = Q$ if $Pred(P) \Leftrightarrow Pred(Q)$. □

Specialisation is a pre-order with bottom $FALSE$ and top $TRUE$ defined as follows.

Definition 5. *(TRUE and FALSE patterns)*
Pattern $TRUE$ is the pattern such that for all models m, $m \models TRUE$. Pattern $FALSE$ is the pattern such that for no model m, $m \models FALSE$. □

In summary, therefore, and letting P, Q and R be any given patterns, we have the following.

$$P \preccurlyeq P \tag{2}$$

$$(P \preccurlyeq Q) \wedge (Q \preccurlyeq R) \Rightarrow (P \preccurlyeq R) \tag{3}$$

$$FALSE \preccurlyeq P \preccurlyeq TRUE \tag{4}$$

3 Operators on Design Patterns

In this section, we review the set of operators on patterns defined in [22]. The restriction operator was first introduced in [3], where it was called the *specialisation* operator.

Definition 6. *(Restriction operator)*
Let P be a given pattern and c be a predicate defined on the components of P. A restriction of P with constraint c, written $P[c]$, is the pattern obtained from P by imposing the predicate c as an additional condition of the pattern. Formally,

1. $Vars(P[c]) = Vars(P)$,
2. $Pred(P[c]) = (Pred(P) \wedge c)$. □

For example, the pattern $Composite_1$ is the variant of the $Composite$ pattern that has only one leaf:

$$Composite_1 = Composite[\#Leaves = 1].$$

Many more examples are given in the case studies reported in [22]. A frequently occuring use is in expressions of the form $P[u = v]$ for pattern P and variables u and v of the same type. This is the pattern obtained from P by unifying components u and v and making them the same element.

The restriction operator does not introduce any new components into the structure of a pattern, but the following operators do.

Definition 7. *(Superposition operator)*
Let P and Q be two patterns. Assume that the component variables of P and Q are disjoint, i.e. $Vars(P) \cap Vars(Q) = \emptyset$. The superposition *of P and Q, written $P * Q$, is defined as follows.*

1. $Vars(P * Q) = Vars(P) \cup Vars(Q)$;
2. $Pred(P * Q) = Pred(P) \wedge Pred(Q)$. □

Informally, $P * Q$ is the minimal pattern (i.e. that with the fewest components and weakest conditions) containing both P and Q without overlap. The definition has the requirement that component variables be disjoint, but we can always systematically rename the variables to make them disjoint and the notation with which we will do so is as follows. Let $x \in Vars(P)$ be a component of pattern P and $x' \notin Vars(P)$. The systematic renaming of x to x' is written as $P[x' := x]$. Obviously, for all models m, we have that $m \models P \Leftrightarrow m \models P[x' := x]$ because $Spec(P)$ is a closed formula. In the sequel, we assume that renaming is made implicitly before two patterns are superposed when there is a naming conflict between them.

Definition 8. *(Extension operator)*
Let P be a pattern, V be a set of variable declarations that are disjoint with P's component variables (i.e. $Vars(P) \cap V = \emptyset$), and c be a predicate with variables in $Vars(P) \cup V$. The extension of pattern P with components V and linkage condition c, written as $P\#(V \bullet c)$, is defined as follows.

1. $Vars(P\#(V \bullet c)) = Vars(P) \cup V$;

2. $Pred(P\#(V \bullet c)) = Pred(P) \wedge c$. □

For any predicate p, let $p[x\backslash e]$ denote the result of replacing all free occurrences of x in p with expression e.

Now we can define the flatten operator as follows.

Definition 9. *(Flatten Operator)*
Let P be a pattern, $xs : \mathbb{P}(T)$ be a variable in $Vars(P)$ and $x : T$ be a variable not in $Vars(P)$. Then the flattening of P on variable x, written $P \Downarrow xs\backslash x$, is defined as follows.

1. $Vars(P \Downarrow xs\backslash x) = (Vars(P) - \{xs : \mathbb{P}(T)\}) \cup \{x : T\}$,
2. $Pred(P \Downarrow xs\backslash x) = Pred(P)[xs\backslash\{x\}]$. □

Note that $\mathbb{P}(T)$ is the power set of T, and thus, $xs : \mathbb{P}(T)$ means that variable xs is a set of elements of type T. For example, $Leaves \subseteq classes$ in the specification of the $Composite$ pattern is the same as $Leaves : \mathbb{P}(classes)$. Applying the flatten operator on $Leaves$, the $Composite_1$ pattern can be equivalently expressed as follows.

$$Composite \Downarrow Leaves\backslash Leaf$$

As an immediate consequence of this definition, we have the following property. For $x_1 \neq x_2$ and $x'_1 \neq x'_2$,

$$(P \Downarrow x_1\backslash x'_1) \Downarrow x_2\backslash x'_2 = (P \Downarrow x_2\backslash x'_2) \Downarrow x_1\backslash x'_1. \quad (5)$$

Therefore, we can overload the $\Downarrow$ operator to a set of component variables. Let X be a subset of P's component variables all of power set type, i.e. $X = \{x_1 : \mathbb{P}(T_1), \cdots, x_n : \mathbb{P}(T_n)\} \subseteq Vars(P), n \geq 1$ and $X' = \{x'_1 : T_1, \cdots, x'_n : T_n\}$ such that $X' \cap Vars(P) = \emptyset$. Then we write $P \Downarrow X\backslash X'$ to denote $P \Downarrow x_1\backslash x'_1 \Downarrow \cdots \Downarrow x_n\backslash x'_n$.

Note that our pattern specifications are closed formulae, containing no free variables. Although the names given to component variables greatly improve readability, they have no effect on semantics so, in the sequel, we will often omit new variable names and write simply $P \Downarrow x$ to represent $P \Downarrow x\backslash x'$. Also, we will use plural forms for the names of lifted variables, e.g. xs for the lifted form of x, and similarly for sets of variables, e.g. XS for the lifted form of X.

Definition 10. *(Generalisation operator)*
Let P be a pattern, $x : T$ be a variable in $Vars(P)$ and $xs : \mathbb{P}(T)$ be a variable not in $Vars(P)$. Then the generalisation *of P on variable x, written $P \Uparrow x\backslash xs$, is defined as follows.*

1. $Vars(P \Uparrow x\backslash xs) = (Vars(P) - \{x : T\}) \cup \{xs : \mathbb{P}(T)\}$,
2. $Pred(P \Uparrow x\backslash xs) = \forall x \in xs \cdot Pred(P)$. □

We will use the same syntactic sugar for $\Uparrow$ as we do for $\Downarrow$. In other words, we will often omit the new variable name and write $P \Uparrow x$, and thanks to an analogue of Equation 5, we can and will promote the operator $\Uparrow$ to sets.

For example, by applying the generalisation operator to $Composite_1$ on the component $Leaf$, we can obtain the pattern $Composite$. Formally,

$$Composite = Composite_1 \Uparrow Leaf \backslash Leaves.$$

The lift operator was first introduced in our previous work [3], but in [22] it is revised so that it only allows lifting class components. Let $CVars(P)$ be the set of variables of patterns P that range over classes, and $OPred(P)$ be the predicate obtained from $Pred(P)$ by the existentially quantifying at the outermost the remaining variables not in $CVars(P)$, i.e. those in $Vars(P) - CVars(P)$, which are the declarations of the operations. Then, we can define lifting as follows.

Definition 11. *(Lift Operator)*
Let P be a pattern and $CVars(P) = \{x_1 : T_1, \cdots, x_n : T_n\}$, $n > 0$. Let $X = \{x_1, \cdots, x_k\}$, $1 \leq k < n$, be a subset of the variables in the pattern. The lifting of P with X as the key, written $P \uparrow X$, is the pattern defined as follows.

1. $Vars(P \uparrow X) = \{xs_1 : \mathbb{P}T_1, \cdots, xs_n : \mathbb{P}T_n\}$,
2. $Pred(P \uparrow X) = \forall x_1 \in xs_1 \cdots \forall x_k \in xs_k \cdot \exists x_{k+1} \in xs_{k+1} \cdots \exists x_n \in xs_n \cdot OPred(P)$. □

When the key set is singleton, we omit the set brackets for simplicity, so we write $P \uparrow x$ instead of $P \uparrow \{x\}$.

For example, $Adapter \uparrow Target$ is the following pattern.

$$Vars(Adapter \uparrow Target) = \{Targets, Adapters, Adaptees \subseteq classes\}$$
$$Pred(Adapter \uparrow Target) = \forall Target \in Targets \cdot \exists Adapter \in Adapter \cdot \exists Adaptee \in Adaptees \cdot OPred(Adapter).$$

Fig. 3 spells out the components and predicates of the pattern.

Specification 3 *(Lifted Object Adapters Pattern)*
Components

1. $Targets, Adapters, Adaptees \subseteq classes$,

Conditions

1. $\forall Adaptee \in Adaptees \cdot \exists specreqs \in Adaptee.opers$,
2. $\forall Target \in Targets \cdot \exists requests \in Target.opers$,
3. $\forall Target \in Targets \cdot CDR(Target)$,
4. $\forall Target \in Targets \cdot \exists Adapter \in Adapters, Adaptee \in Adaptees \cdot$
   (a) $Adapter \longrightarrow\!\!\triangleright\, Target$,
   (b) $Adapter \longrightarrow Adaptee$,
   (c) $\forall o \in Target.requests \cdot \exists o' \in Adaptee.specreqs \cdot (calls(o, o')))$

Fig. 3. Specification of Lifted Object Adapter Pattern

Informally, lifting a pattern P results in a pattern P' that contains a number of instances of P. For example, $Adapter \uparrow Target$ is the pattern that contains a number of

$Targets$ of adapted classes. Each of these has a dependent $Adapter$ and $Adaptee$ class configured as in the original $Adapter$ pattern. In other words, the component $Target$ in the lifted pattern plays a role similar to the *primary key* in a relational database.

4 Algebraic Laws of the Operations

This section studies the algebraic laws that the operators obey. For the sake of space, we only give some proofs in the appendix.

4.1 Laws of Restriction

Let $vars(p)$ denote the set of free variables in a predicate p. For all predicates c, c_1, c_2 such that $vars(c), vars(c_1)$ and $vars(c_2) \subseteq Vars(P)$, the following equalities hold.

$$P[c_1] \preccurlyeq P[c_2],\ if\ c_1 \Rightarrow c_2 \quad (6)$$

$$P[c] \preccurlyeq P[true] \quad (7)$$

$$P[c][c] = P[c] \quad (8)$$

$$P[c_1][c_2] = P[c_2][c_1] \quad (9)$$

$$P[c_1][c_2] = P[c_1 \wedge c_2] \quad (10)$$

$$P[true] = P \quad (11)$$

$$P[false] = FALSE \quad (12)$$

4.2 Laws of Superposition

For all patterns P and Q, we have the following equations.

$$P * Q \preccurlyeq P \quad (13)$$

$$Q \preccurlyeq P \Rightarrow P * Q = Q \quad (14)$$

From this and reflexivity of $\preccurlyeq$, it follows that superposition is idempotent.

$$P * P = P \quad (15)$$

It also follows from (14) that $TRUE$ is the unit of superposition since it is the top in $\preccurlyeq$. Similarly, $FALSE$ is the zero of superposition since it is the bottom in $\preccurlyeq$.

$$P * TRUE = TRUE * P = P \quad (16)$$

$$P * FALSE = FALSE * P = FALSE \quad (17)$$

Superposition is also commutative and associative.

$$P * Q = Q * P \quad (18)$$

$$(P * Q) * R = P * (Q * R) \quad (19)$$

4.3 Laws of Extension

The extension operation has the following properties.

Let U be any set of component variables that is disjoint to $Vars(P)$, and c_1, c_2 be any given predicates such that $vars(c_i) \subseteq Vars(P) \cup U$, $i = 1, 2$. We have the following inequalities.

$$P\#(U \bullet c_1) \preccurlyeq P\#(U \bullet c_2),\ if\ c_1 \Rightarrow c_2 \tag{20}$$

$$P\#(U \bullet c_1) \preccurlyeq P \tag{21}$$

Let U and V be any sets of component variables that are disjoint to $Vars(P)$ and to each other, c_1 and c_2 be any given predicates such that $vars(c_1) \subseteq Vars(P) \cup U$ and $vars(c_2) \subseteq Vars(P) \cup V$. We have equalities.

$$P\#(U \bullet c_1)\#(V \bullet c_2) = P\#(U \cup V \bullet c_1 \wedge c_2) \tag{22}$$

$$P\#(U \bullet c_1)\#(V \bullet c_2) = P\#(V \bullet c_2)\#(U \bullet c_1) \tag{23}$$

4.4 Laws of Flattening and Generalisation

Let $X, Y \subseteq Vars(P)$ and $X \cap Y = \emptyset$.

$$(P \Downarrow X) \Downarrow Y = P \Downarrow (X \cup Y) \tag{24}$$

$$(P \Uparrow X) \Uparrow Y = P \Uparrow (X \cup Y) \tag{25}$$

4.5 Laws Connecting Several Operators

For all predicates c such that $vars(c) \subseteq Vars(P)$, we have that

$$P[c] * Q = (P * Q)[c]. \tag{26}$$

For all $X \subseteq Vars(P)$, we have that

$$(P \Uparrow X) * Q = (P * Q) \Uparrow X, \tag{27}$$

$$(P \Downarrow X) * Q = (P * Q) \Downarrow X. \tag{28}$$

Let $X \subseteq Vars(P) \cup Vars(Q)$. From (24), (27) and (28), we can prove that

$$(P * Q) \Uparrow X = (P \Uparrow X_P) * (Q \Uparrow X_Q), \tag{29}$$

$$(P * Q) \Downarrow X = (P \Downarrow X_P) * (Q \Downarrow X_Q), \tag{30}$$

where $X_P = X \cap Vars(P)$, $X_Q = X \cap Vars(Q)$.

For all sets of variables X such that $X \cap vars(P) = \emptyset$ and all predicates c such that $Vars(c) \subseteq (Vars(P) \cup X)$, we have that

$$P\#(X \bullet c) = P\#(X \bullet True)[c]. \tag{31}$$

$$P\#(X \bullet c) = P[\exists X \cdot c], \tag{32}$$

where $\exists X \cdot c = \exists x_1 : T_1 \cdots \exists x_k : T_k \cdot c$, if $X = \{x_1 : T_1, \cdots, x_k : T_k\}$.

For all $x \in Vars(P)$ such that $x : \mathbb{P}(T)$, we have that

$$P \Downarrow (x \backslash x') = P\#(\{x' : T\} \bullet (x = \{x'\}). \tag{33}$$

For all $X \subseteq Vars(P)$ and $X' \cap Vars(P) = \emptyset$, we have that

$$P \Uparrow X \backslash X' = (P \uparrow X \backslash X') \Downarrow (V - X'), \tag{34}$$

$$(P \Uparrow X \backslash X') \Downarrow (X' \backslash X) = P. \tag{35}$$

where $V = Vars(P \uparrow X)$.

From (34) and (35), we can prove that for all $x \in Vars(P)$,

$$(P \uparrow x) \Downarrow V = P, \tag{36}$$

where $V = Vars(P \uparrow x)$.

Let $X \subseteq Vars(P)$, we have that

$$(P \uparrow X) * Q = ((P * Q) \uparrow X) \Downarrow Vars(Q). \tag{37}$$

Let c be a predicate that $vars(c) \subseteq X \cup V \subseteq Vars(P)$, we have that

$$((P[c] \uparrow X) \Downarrow VS) = ((P \uparrow X) \Downarrow VS)[c'], \tag{38}$$

where $c' = \forall x_1 : xs_1, \cdots, \forall x_k : xs_k \cdot c$, $\{x_1, \cdots, x_k\} = vars(c) \cap X$.

5 Examples

In this section, we demonstrate the uses of the laws to prove the equivalence of pattern compositions.

We first consider the composition of $Composite$ and $Adapter$ in such a way that one of the $Leaves$ in the $Composite$ pattern is the $Target$ in the $Adapter$ pattern. This leaf is renamed as the $AdaptedLeaf$. The definition for the composition using the operators is as follows:

$$\begin{array}{l} OneAdaptedLeaf \triangleq \\ \quad (Adapter * Composite)[Target \in Leaves][AdaptedLeaf := Target] \end{array}$$

Then, we lifted the adapted leaf to enable several of these $Leaves$ to be adapted. That is, we lift the $OneAdaptedLeaf$ pattern with $AdaptedLeaf$ as the key and then flatten those components in the composite part of the pattern (i.e. the components in the $Composite$ pattern remain unchanged). Formally, this is defined as follows.

$$\begin{array}{l} (OneAdaptedLeaf \uparrow (AdaptedLeaf \backslash AdaptedLeaves)) \\ \quad \Downarrow \{Composites, Components, Leaveses\} \end{array} \tag{39}$$

By the definitions of the operators, we derive the predicates of the pattern in Specification 4 after some simplification in the first order logic.

Specification 4 *(ManyAdaptedLeaves)*
Components

1. $Component, Composite \in classes$,
2. $Leaves, AdaptedLeaves, Adapters, Adaptees \subseteq classes$,
3. $ops \subseteq Component.opers$

Static Conditions

1. $ops \neq \emptyset$
2. $\forall o \in ops.isAbstract(o)$,
3. $\forall l \in Leaves.(l \longrightarrow\!\!\triangleright^{+} Component \wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
4. $\forall l \in AdaptedLeaves.(l \longrightarrow\!\!\triangleright^{+} Component \wedge \neg(l \diamond\!\!\longrightarrow^{+} Component))$
5. $isInterface(Component)$,
6. $Composite \longrightarrow\!\!\triangleright^{+} Component$
7. $Composite \diamond\!\!\longrightarrow^{*} Component$
8. $CDR(Component)$
9. $\forall Adaptee \in Adaptees \cdot (\exists specreqs \in Adaptee.opers$,
10. $\forall AdLeaf \in AdaptedLeaves \cdot \exists requests \in AdLeaf.opers$,

Dynamic Conditions

1. *any call to* $Composite$ *causes follow-up calls*
$$\forall m \in messages \cdot \exists o \in ops \cdot (toClass(m) = Composite \wedge m.sig \approx o \Rightarrow \exists m' \in messages \,.\, calls(m, m') \wedge m'.sig \approx m.sig)$$
2. *any call to a leaf or an adapted leaf does not*
$$\forall m \in messages \cdot (\exists o \in ops \cdot (toClass(m) \in Leaves \cup AdaptedLeaves \wedge m.sig \approx o) \Rightarrow \neg\exists m' \in messages \,.\, calls(m, m') \wedge m'.sig \approx m.sig))$$
3. $\forall AdLeaf \in AdaptedLeaves \cdot \exists Adapter \in Adapters, Adaptee \in Adaptees \cdot$
   (a) $Adapter \longrightarrow\!\!\triangleright AdLeaf$,
   (b) $Adapter \longrightarrow Adaptee$,
   (c) $\forall o \in AdLeaf.requests \cdot \exists o' \in Adaptee.specreqs \cdot (calls(o, o')))$

An alternative way of expressing the composition is first to lift the $Adapter$ with $target$ as the key and then to superposition it to the $Composite$ patterns so that many leaves can be adapted. Formally,

$$ManyAdaptedLeaves \triangleq (((Adapter \uparrow (Target\backslash Targets)) * Composite)[Targets \subseteq Leaves] [AdaptedLeaves := Targets]$$

We now apply the algebraic laws to prove that expression Equ. (39) is equivalent to the definition of $ManyAdaptedLeaves$.

First, by (37), we can rewrite $ManyAdaptedLeaves$ to the following expression, where $V_C = \{Composites, Components, Leaveses\}$.

$$((Adapter * Composite) \uparrow (Target\backslash Targets) \Downarrow V_C [Targets \subseteq Leaves] [Adaptedleaves := Targets] \quad (40)$$

Because $Leavses$ is in V_C and $Targets \subseteq Leaves$ is equivalent to

$$\forall Target \in Targets \cdot (Target \in Leaves),$$

by (38), we have that

$$((Adapter * Composite) \uparrow (Target \backslash Targets) \Downarrow V_C)\, [Targets \subseteq Leaves] \quad (41)$$
$$= ((Adapter * Composite)[Target \in Leaves]) \uparrow (Target \backslash Targets) \Downarrow V_C$$

Now, renaming $Target$ to $AdaptedLeaf$ and $Targets$ to $AdaptedLeaves$ in expression on the right-hand-side of (41), we have the following.

$$((Adapter * Composite)[Target \in Leaves][AdaptedLeaf := Target])$$
$$\uparrow (AdaptedLeaf \backslash AdaptedLeaves) \Downarrow V_C \quad (42)$$

By substituting the definition of $OneAdaptedLeaf$ into Equ. (42), we obtain (39).

6 Conclusion

In this paper, we proved a set of algebraic laws that the operators on design patterns obey and we demonstrated their use in proving the equivalence of pattern compositions. These operators and algebraic laws form a formal calculus of design patterns that enable us to reasoning about pattern compositions. Although the calculus is developed in our own formalisation framework, we believe that they can be easily adapted to others, such as that of Eden's approach, which also uses first-order logic but no specification of behavioural features [10], that of Taibi's approach, which is a mixture of first-order logic and temporal logic [19], and that of [12], etc. as well as the approaches based on graphic meta-modelling languages, such as RBML [8] and DPML [14]. However, the definitions of the operators and proofs of the laws are more concise and readable in our formalism.

For future work, we are investigating the uses of theorem provers for automated reasoning about the compositions of design patterns based on the theory developed in this paper. We are also investigating the completeness of the algebraic laws.

References

1. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies, 2nd edn. Prentice Hall, Englewood Cliffs (2003)
2. Bayley, I., Zhu, H.: Formalising design patterns in predicate logic. In: Proc. of SEFM 2007, pp. 25–36. IEEE Computer Society, Los Alamitos (2007)
3. Bayley, I., Zhu, H.: On the composition of design patterns. In: Proc. of QSIC 2008, pp. 27–36. IEEE Computer Society, Los Alamitos (2008)
4. Bayley, I., Zhu, H.: Specifying behavioural features of design patterns in first order logic. In: Proc. of COMPSAC 2008, pp. 203–210. IEEE Computer Society, Los Alamitos (2008)
5. Bayley, I., Zhu, H.: Formal specification of the variants and behavioural features of design patterns. Journal of Systems and Software 83(2), 209–221 (2010)
6. Buschmann, F., Henney, K., Schmidt, D.C.: Past, present, and future trends in software patterns. IEEE Software 24(4), 31–37 (2007)

7. Eden, A.H.: Formal specification of object-oriented design. In: International Conference on Multidisciplinary Design in Engineering, Montreal, Canada (November 2001)
8. France, R.B., Kim, D.K., Ghosh, S., Song, E.: A UML-based pattern specification technique. IEEE Trans. Softw. Eng. 30(3), 193–206 (2004)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
10. Gasparis, E., Nicholson, J., Eden, A.H.: LePUS3: An object-oriented design description language. In: Stapleton, G., Howse, J., Lee, J. (eds.) Diagrams 2008. LNCS (LNAI), vol. 5223, pp. 364–367. Springer, Heidelberg (2008)
11. Hou, D., Hoover, H.J.: Using SCL to specify and check design intent in source code. IEEE Trans. Softw. Eng. 32(6), 404–423 (2006)
12. Lano, K., Bicarregui, J.C., Goldsack, S.: Formalising design patterns. In: BCS-FACS Northern Formal Methods Workshop, Ilkley, UK (September 1996)
13. Lauder, A., Kent, S.: Precise visual specification of design patterns. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 114–134. Springer, Heidelberg (1998)
14. Mapelsden, D., Hosking, J., Grundy, J.: Design pattern modelling and instantiation using DPML. In: Proc. of CRPIT 2002, pp. 3–11. Australian Computer Society, Inc. (2002)
15. Mikkonen, T.: Formalizing design patterns. In: Proc. of ICSE 1998, pp. 115–124. IEEE CS, Los Alamitos (April 1998)
16. Nija Shi, N., Olsson, R.: Reverse engineering of design patterns from Java source code. In: Proc. of ASE 2006, pp. 123–134 (September 2006)
17. Riehle, D.: Composite design patterns. In: Proc. of OOPSLA 1997, pp. 218–228 (1997)
18. Taibi, T.: Formalising design patterns composition. Software, IEE Proceedings 153(3), 126–153 (2006)
19. Taibi, T., Check, D., Ngo, L.: Formal specification of design patterns-a balanced approach. Journal of Object Technology 2(4) (July-August 2003)
20. Zhu, H., Shan, L.: Well-formedness, consistency and completeness of graphic models. In: Proc. of UKSIM 2006, pp. 47–53 (April 2006)
21. Zhu, H.: On the theoretical foundation of meta-modelling in graphically extended BNF and first order logic. In: Proc. of TASE 2010. IEEE CS Press, Taipei (August 2010)
22. Zhu, H., Bayley, I.: A formal language of pattern composition. In: Proc. of PATTERNS 2010, Lisbon, Portugal (November 2010) (in Press)
23. Zhu, H., Bayley, I., Shan, L., Amphlett, R.: Tool support for design pattern recognition at model level. In: Proc. of COMPSAC 2009, pp. 228–233. IEEE CS, Los Alamitos (July 2009)
24. Zhu, H., Shan, L., Bayley, I., Amphlett, R.: A formal descriptive semantics of UML and its applications. In: Lano, K. (ed.) UML 2 Semantics and Applications. John Wiley & Sons, Inc., Chichester (November 2009)

Appendix: Proofs of the Algebraic Laws

In this appendix, we give some proofs of the algebraic laws.
Proof of Laws of Restriction:

For Law (6), let P be any given pattern, and c_1, c_2 be any predicates such that $vars(c_i) \subseteq Vars(P)$, $i = 1, 2$. By Definition 6, we have $Vars(P[c_i]) = Vars(P)$, and $Pred(P[c_i]) = Pred(P) \wedge c_i$, for $i = 1, 2$. Assume that $c_1 \Rightarrow c_2$. Then, we have that $Pred(P[c_1]) = Pred(P) \wedge c_1 \Rightarrow Pred(P) \wedge c_2 \equiv Pred(P[c_2])$. So by Lemma 4, we have that $P[c_1] \preccurlyeq P[c_2]$.

Similarly, we can prove that $Pred(P[true]) \equiv Pred(P)$ and $Pred(P[c_1][c_2]) \equiv Pred(P[c_1 \wedge c_2]$, thus, Law (10) and (11) are true by Lemma 4.

Law (7) is the special case of (6) where c_2 is $true$. For (8), we have that $c \wedge c \equiv c$. Thus, it follows from (10).

Law (12) holds because $Pred(P[false])$ cannot be satisfied by any models. □

For the majority of laws, the variable sets on the two sides of the law can be proven to be equal. Therefore, by Lemma 4, the proof of the law reduces to the proof of the equivalence or implication between the predicates. However, for some laws, these variable sets are not equal. In such cases, we use Lemma 3. The following is an example of such a proof.

Proof of Law (13)

Let P and Q be patterns with

$$Vars(P) = \{x_1, \dots, x_m\}, Vars(Q) = \{y_1, \dots, y_n\}.$$

Assume that

$$Vars(P) \cap Vars(Q) = \emptyset. \tag{43}$$

$$\begin{array}{ll} Spec(P * Q) & \\ = \exists x_1, \dots x_m, y_1 \dots y_n \cdot Pred(P) \wedge Pred(Q), & < Def. 1 > \\ \equiv \exists x_1, \dots, x_m \cdot Pred(P) \wedge \exists y_1 \dots y_n \cdot Pred(Q), & < (43) > \\ \Rightarrow \exists x_1, \dots, x_m \cdot Pred(P), & < FOL > \\ = Spec(P), & < Def. 1 > \end{array}$$

Thus, by Lemma 3, we have that $(P * Q) \preccurlyeq P$. □

The following of the proof of Law (38), which involves three operators.

Proof of Law (38):

First, we prove that the variable sets on the two sides of the equation are equal.

Let $Y = Vars(P) - (X \cup V)$. Then, we have that $Vars(P) = X \cup Y \cup V$. By definition of the operators, it is easy to see that

$$Vars(lhs) = (((XS \cup YS \cup VS) - VS) \cup V) = (XS \cup YS \cup V) = Vars(rhs).$$

Thus, we only need to prove the predicates of the two sides are equivalent. Let $X = \{x_1, \cdots, x_k\}$, $Y = \{y_1, \cdots, y_n\}$ and $V = \{v_1, \cdots, v_m\}$.

By the definitions of the operators, we have that $Pred(lhs)$ is

$$\begin{array}{l} \forall x_1 \in xs_1 \dots x_k \in xs_k \cdot \exists y_1 \in ys_1 \dots y_n \in ys_n \cdot \\ \quad \exists v_1 \in vs_1 \dots v_m \in vs_m \cdot (Pred(P) \wedge c)[vs_1 \backslash \{v_1\}] \dots [vs_m \backslash \{v_m\}] \\ \equiv \forall x_1 \in xs_1 \dots x_k \in xs_k \cdot \exists y_1 \in ys_1 \dots y_n \in ys_n \cdot \\ \quad \exists v_1 \in vs_1 \dots v_m \in vs_m \cdot (Pred(P)[vs_1 \backslash \{v_1\}] \dots [vs_m \backslash \{v_m\}] \\ \quad \wedge c[vs_1 \backslash \{v_1\}] \dots [vs_m \backslash \{v_m\}]) \\ \equiv \forall x_1 \in xs_1 \dots x_k \in xs_k \cdot \exists y_1 \in ys_1 \dots y_n \in ys_n \cdot (Pred(P) \wedge c) \end{array}$$

Because $vars(c) \cap Y = \emptyset$, the above is equivalent to the following.

$$\begin{array}{l} \forall x_1 \in xs_1 \dots x_k \in xs_k \cdot \exists y_1 \in ys_1 \dots y_n \in ys_n \cdot Pred(P) \\ \quad \wedge \forall x_1 \in xs_1 \dots x_k \in xs_k \cdot c \end{array}$$

This is $Pred(rhs)$. By Lemma 4, the law holds. □

Dynamic Resource Reallocation between Deployment Components*

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

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

Abstract. Today's software systems are becoming increasingly configurable and designed for deployment on a plethora of architectures, ranging from sequential machines via multicore and distributed architectures to the cloud. Examples of such systems are found in, e.g., software product lines, service-oriented computing, information systems, embedded systems, operating systems, and telephony. To model and analyze systems without a fixed architecture, the models need to naturally capture and range over relevant deployment scenarios. For this purpose, it is interesting to lift aspects of low-level deployment concerns to the abstraction level of the modeling language. In this paper, the object-oriented modeling language Creol is extended with a notion of dynamic deployment components with parametric processing resources, such that processor resources may be explicitly reallocated. The approach is compositional in the sense that functional models and reallocation strategies are both expressed in Creol, and functional models can be run alone or in combination with different reallocation strategies. The formal semantics of deployment components is given in rewriting logic, extending the semantics of Creol, and executes on Maude, which allows simulations and test suites to be applied to models which vary in their available resources as well as in their resource reallocation strategies.

1 Introduction

Software systems today are increasingly being developed to be highly configurable, not only with respect to the functionality provided by a specific instance of the system but also with respect to the targeted deployment architecture. An example of a development method which attempts to systematize this variability, is software product line engineering [23]; in a product line, different software systems (or products) may be instantiated with different features and for different architectures. Deployment variability may be found in operating systems, which can be adapted to specific hardware and even to different numbers of available kernels; web shops, which are deployed on a varying number of servers and may even dynamically perform load balancing between these servers; and information systems within, e.g., healthcare or finance, which may run on a single computer,

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

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 646–661, 2010.
© Springer-Verlag Berlin Heidelberg 2010

in a distributed set-up, or even in the cloud. Software product lines raise new challenges for the performance analysis of component-based applications [27]. In this paper, we consider the performance analysis of object-oriented component or system models in deployment scenarios where the amount of processing resources available to a component may vary over time.

Our work is based on Creol [10,17], a modeling language for concurrent objects communicating by asynchronous method calls. Creol has an operational semantics in rewriting logic [21] which is executable on Maude [9]. Concurrent objects resemble Actors [2] and Erlang [4] processes: Objects are inherently concurrent, conceptually each object has a dedicated processor, and there is at most one activity in an object at any time. This concurrency model is attracting attention as an alternative to multi-thread concurrency in object-orientation (e.g., [6]), and been integrated with, e.g., Java [26] and Scala [13]. Concurrent objects support compositional verification of concurrent software [3, 10], in contrast to multi-threading [1]. A distinguishing feature of Creol is its cooperative scheduling of method activations inside concurrent objects. Recently, Creol's notion of cooperative scheduling and asynchronous method calls has been integrated in Java by means of concurrent object groups [24].

This paper generalizes the idea of concurrent object groups to *dynamic deployment components* which are parametric in the amount of concurrent activity they allow within a time interval, and between which resources may be reallocated. Creol is extended with notions of timed execution and deployment components, which are integrated into Creol's operational semantics. This integration is non-trivial in that it must capture parametric concurrent activities within time intervals in terms of an interleaving concurrency semantics in order to execute the models on Maude. Deployment scenarios varying in the resources available to the deployment components, may be validated by means of test suites, executed on Maude. This allows the timed behavior of concurrent object models under restricted concurrency assumptions, as well as load balancing and process migration strategies between components, to be validated and compared.

Paper overview. Sect. 2 presents a timed version of Creol and Sect. 3 the dynamic deployment components. Sect. 4 illustrates this extension by the modeling and simulation of an example. Sect. 5 explains the operational semantics of the extended language and Sect. 6 discusses related work, and Sect. 7 concludes.

2 Concurrent Objects in Creol

Creol is an abstract behavioral modeling language for distributed active objects, based on asynchronous method calls and processor release points. In Creol, objects conceptually have dedicated processors and live in a distributed environment with asynchronous and unordered communication between objects. Communication is between named objects by means of asynchronous method calls; these may be seen as triggers of concurrent activity, resulting in new activities (so-called *processes*) in the called object. This section briefly introduces Creol (for further details see, e.g., [10,17]). Objects are dynamically created instances of

Syntactic categories.	*Definitions.*
C, I, m in Names	$IF ::= \textbf{interface}\ I\ \{\ [\overline{Sg}]\ \}$
g in Guard	$CL ::= \textbf{class}\ C\ [(\overline{I\ x})]\ [\textbf{implements}\ \overline{I}]\ \{\ [\overline{I\ x;}]\ \overline{M}\}$
s in Stmt	$Sg ::= I\ m\ ([\overline{I\ x}])$
x in Var	$M ::= Sg == [\overline{I\ x;}]\ \{\ s\ \}$
e in Expr	$g ::= b \mid x? \mid g \wedge g$
b in BoolExpr	$s ::= s; s \mid x := rhs \mid \textbf{release} \mid \textbf{await}\ g \mid \textbf{return}\ e$
	$\mid \textbf{if}\ b\ \textbf{then}\ \{\ s\ \}\ [\textbf{else}\ \{\ s\ \}] \mid \textbf{while}\ b\ \{\ s\ \} \mid \textbf{skip}$
	$e ::= x \mid b \mid \textbf{this} \mid \textbf{now} \mid \textbf{null}$
	$rhs ::= e \mid \textbf{new}\ C(\overline{e}) \mid [e]!m(\overline{e}) \mid [e.]m(\overline{e}) \mid x.\textbf{get}$

Fig. 1. The syntax of core Timed Creol. Terms such as $\overline{e}$ and $\overline{x}$ denote lists over the corresponding syntactic categories, square brackets [ ] denote optional elements. Expressions e and guards g are side-effect free; Boolean expressions b include comparison by means of equality, greater- and less-than operators. Expressions on other datatypes (strings, numbers) are written in the usual way and not contained in this figure.

classes, declared attributes are initialized to some arbitrary type-correct values. An optional *init* method may be used to redefine the attributes. Active behavior, triggered by an optional *run* method, is interleaved with passive behavior, triggered by method calls. Thus, an object has a set of processes to be executed, which stem from method activations. Among these, at most one process is *active* and the others are *suspended* on a process queue. Process scheduling is by default non-deterministic, but controlled by *processor release points* in a cooperative way. Creol is strongly typed: for well-typed programs, invoked methods are supported by the called object (when not *null*), such that formal and actual parameters match. This paper assumes that programs are well-typed.

Figure 1 gives the syntax for a core subset of Timed Creol (omitting, e.g., inheritance). A *program* consists of interface and class definitions and a `main` method to configure the initial state. *IF* defines an interface with name I and method signatures Sg. A class implements a list $\overline{I}$ of interfaces, specifying types for its instances. CL defines a class with name C, interfaces $\overline{I}$, class parameters and state variables x (of type I), and methods M. (The *attributes* of the class are both its parameters and state variables.) A method signature Sg declares the return type I of a method with name m and formal parameters $\overline{x}$ of types $\overline{I}$. M defines a method with signature Sg, a list of local variable declarations $\overline{x}$ of types $\overline{I}$, and a statement s. Statements may access class attributes, locally defined variables, and the method's formal parameters.

Statements. Assignment $x := rhs$, sequential composition $s_1; s_2$, and **if**, **skip**, **while**, and **return** constructs are standard. The statement **release** unconditionally releases the processor by suspending the active process. In contrast, the guard g controls processor release in the statement **await** g, and consists of Boolean conditions b and return tests $x?$ (see below). If g evaluates to false, the current process is *suspended* and the execution thread becomes idle. When the execution thread is idle, any enabled process from the pool of suspended processes may be scheduled. Explicit signaling is therefore redundant.

Expressions rhs include declared variables x, object identifiers o, Boolean expressions b, and object creation **new** $C(\overline{e})$ and **null**. The specially reserved read-only variable **this** refers to the identifier of the object and **now** refers to the current clock value (explained below). Note that pure expressions are denoted by e and that remote access to attributes is not allowed. (The full language includes a functional expression language with standard operators for data types such as strings integers lists, sets, maps, and tuples. These are omitted in the core syntax, and explained when used in the examples.)

Communication in Creol is based on asynchronous method calls, denoted $o!m(\overline{e})$, and future variables. (Local calls are written $!m(\overline{e})$.) After making an asynchronous call $x := o!m(\overline{e})$, the caller may proceed with its execution without blocking on the call. Here x is a future variable, o is an object expression, and $\overline{e}$ are (data value or object) expressions. A future variable x refers to a return value which has yet to be computed. There are two operations on future variables, controlling external synchronization in Creol. First, the guard **await** $x?$ suspends the active process unless a return to the call associated with x has arrived (allowing other processes in the object to be scheduled). Second, the return value is retrieved by the expression x.**get**, which blocks all execution in the object until the return value is available. The statement sequence $x := o!m(\overline{e});\ v := x$.**get** encodes a *blocking call*, abbreviated $v := o.m(\overline{e})$ (often referred to as a synchronous call), whereas the statement sequence $x := o!m(\overline{e});$ **await** $x?;\ v := x$.**get** encodes a non-blocking, *preemptable call*.

Time. We consider a discrete time model, comparable to a system clock which updates every n milliseconds. With this granularity of time, an object which executes a statement may, but need not observe that time has advanced. The expression **now** returns the present time, i.e., the global clock's value in the current state. Time values are totally ordered by the less-than operator; comparing two time values result in a Boolean value which may be used as a guard in **await** statements. From an object's local perspective the passage of time is indirectly observable; time can advance by either evaluating statements, blocking, or simply awaiting the passage of time. This model of time combined with Creol's blocking and non-blocking synchronization semantics, is powerful enough to express both process- and object-wide *progress* statements.

3 Dynamic Deployment Components

Creol's object model is inherently concurrent, which means that for the actual deployment of a program it is necessary to map the logical concurrency of the model to physical computing resources. For this purpose, we introduce a notion of *deployment component* into the modeling language, which abstracts from the number and speed of the physical processors available to the component by a notion of *concurrent resource*. The granularity of the global time model defines the points in time when the executing system is observable. Concurrent resources may be consumed in parallel or in sequential order, which reflects the number of processors and their speeds relative to the granularity of the time intervals

$$s ::= \ldots \mid \textbf{transfer}(e, e)$$
$$e ::= \ldots \mid \textbf{mycomp} \mid \textbf{available} \mid \textbf{load}(e)$$
$$rhs ::= \ldots \mid \textbf{component}(r) \mid \textbf{new}\ C\,(\overline{e})\ \textbf{in}\ e$$

Fig. 2. Extension of the syntax for deployment components (r in Resources) in Fig. 1

of the model. Thus, the logical concurrency model of the concurrent objects is controlled by their associated deployment component. A deployment component is parametric in the computational resources it offers to a group of dynamically created objects, which allows easy configuration of concurrent resources.

The execution inside a deployment component can be understood as follows. Let n be a natural number. Resources are modeled by a data type `Resource` which extends the natural numbers with an "unlimited resource" ω, such that resource consumption is captured by subtraction, where $\omega - n = \omega$. Within a time interval, a deployment component with r concurrent resources may execute up to n execution steps in parallel, where $n \leq r$. Consider a deployment component D instantiated with r resources and let G be the set of concurrent objects which currently reside in the deployment component. Let $A \subseteq G$ be a subset of the concurrent objects on the component, such that objects in A are able to perform an execution step in their current state. Provided $|A| \leq r$, every object in A may consume a resource, leaving $r' = r - |A|$ resources available on the component. If there are remaining resources (i.e., $r' > 0$) , another set of execution steps is performed if possible within the same time interval by repeating this procedure.

In the modeling language, an object exists in the context of a deployment component with a given amount of resources, and may have variables x of type `Component` which refer to deployment components. A new deployment component is created by the statement x:=**component**(r), which allocates a given quantity of concurrent resources r to the component x (capturing the actual processing capacity of x) by correspondingly reducing the resources of the current deployment component. The set of concurrent objects residing on the components, representing the logically concurrent activities, may grow dynamically. When objects are created, they must reside inside a deployment component. The syntax for object creation is extended with an optional clause to specify the targeted deployment component in the expression **new** $C(\overline{e})$ **in** x. This expresses that the new C object will reside in the component x. Objects generated by a parent object residing in a component x will also reside in x unless otherwise specified by an **in** clause. Thus the behavior of a Creol model which does not statically declare additional deployment components can be captured by a root deployment component with ω resources.

In the context of a given deployment component dc, the expression **mycomp** returns dc, **available** returns the number of resources currently allocated to dc, and **load**(e) returns the average number of used resources in dc during the last e time intervals. The statement **transfer**(x, r) reallocates r resources from dc to a component x. The language extension is summarized in Fig. 2.

4 Example: Phone Services during New Year's Eve

At midnight on new year's eve the behavior of cellphone users briefly changes from normal usage (i.e., a fairly low number of calls and messages) to sending large numbers of SMS messages. We use this phenomenon to motivate and illustrate resource reallocation by means of two cooperating deployment components. The model consists of two services, `TelephoneService` and `SMSService`, and a number of handset clients interacting with either the telephony or

```
1  interface TelephoneService { Void call(Int duration); }
2
3  interface SMSService { Void sendSMS(); }
4
5  class TelephoneService implements TelephoneService {
6      Void call(Int duration) {
7          Time t; t := now;
8          await now >= t + duration;
9      }
10 }
11 class SMSService implements SMSService {
12     Void sendSMS() { skip; }
13 }
```

Fig. 3. Creol interfaces and classes for the telephony and SMS services

```
1  class Handset (Int cycle, TelephoneService ts, SMSService  smss) {
2      Time created := now;
3      Bool call := false;
4
5      Void normalBehavior() {
6          Time t := now;
7          if (now > created + 50 ∧ now < created + 70) {
8              !midnightWindow();
9          } else {
10             if (call) ts.call(1;) else smss!sendSMS()
11             call := ¬call;
12             await now >= t + cycle;
13             !normalBehavior(); } }
14
15     Void midnightWindow() {
16         Time t := now;
17         Int i := 0;
18         if (now > created + 70) {
19             !normalBehavior();
20         } else {
21             while (i < 10) { smss!sendSMS(); i := i+1; }
22             await now > t;
23             !midnightWindow(); } }
24
25     op run() { !normalBehavior(); }
26 }
```

Fig. 4. The `Handset` class, implementing "Happy New Year" behavior. Before and after midnight, users alternate between short calls and sending single messages. During the midnight window ($50 \leq t \leq 70$), ten SMS per interval are sent.

```
1  interface Balancer { Void setPartner(Balancer p);
   Void request(Component comp); }
2
3  class Balancer {
4      Balancer partner := null;
5      Void run () {
6          Time t := now;
7          await now > t;
8          if (partner≠null ∧ available<load(1)*0.9) {
9              partner.request(mycomp);}
10         !run(); }
11     Void request(Component comp) {
12         if (load(1) < available−10) {transfer(comp,available/2);} }
13     Void setPartner(Balancer p) { partner := p; }
14 }
15
16 Void main() {
17     Component smscomp := component(50);
18     Component telcomp := component(50);
19     SMSService sms := new SMSService() in smscomp;
20     TelephoneService tel := new TelephoneService() in telcomp;
21     Balancer smsb := new Balancer in smscomp;
22     Balancer telb := new Balancer in telcomp;
23     smsb.setPartner(telb);    telb.setPartner(smsb);
24     Client c := new Handset(1,tel,sms); c := new Handset(1,tel,sms);
25     c := new Handset(1,tel,sms); c := new Handset(1,tel,sms);
26 }
```

Fig. 5. A resource reallocation strategy and deployment configuration. Lines 21-23 initiate resource balancing; without these lines, the model runs with no functional changes but it has a different timing behavior due to overload in the SMS deployment component. Since `Handset` objects are active, references to them are not needed.

messaging service. The interfaces and implementations of the two services are given in Fig. 3. The method `call` will be called synchronously; as a parameter the client provides a duration for the call. The method `sendSMS` will be called asynchronously. Note that this model abstracts from many details (e.g. data model, bandwidth, server internals), which can be added as needed. The model of the handset clients interoperating with the services is given in Fig. 4. Client behavior is regulated by a parameter `cycle`, which determines the frequency of phone calls and messages of the handset. Between time $t = 50$ and 70, `Handset` objects change behavior and send SMS messages in a rapid pace.

Simulating this model in a scenario with ω resources leads to a *purely behavioral model*, in which each object acts according to its specification (as in normal Creol). Placing the SMS service in an environment with restricted resources will lead to observable overload during the midnight window, given sufficient clients to consume all its resources. (Recall that the load history of a deployment component over time can be extracted from a simulation run via its `Load` attribute.)

The proposed resource-related language constructs (i.e., **`available`**, **`load`**, and **`transfer`**) allow different load balancing schemes to be expressed. In Fig. 5 the `main` method defines an example scenario where each service runs in its own deployment component, created with 50 resources, and three client objects run

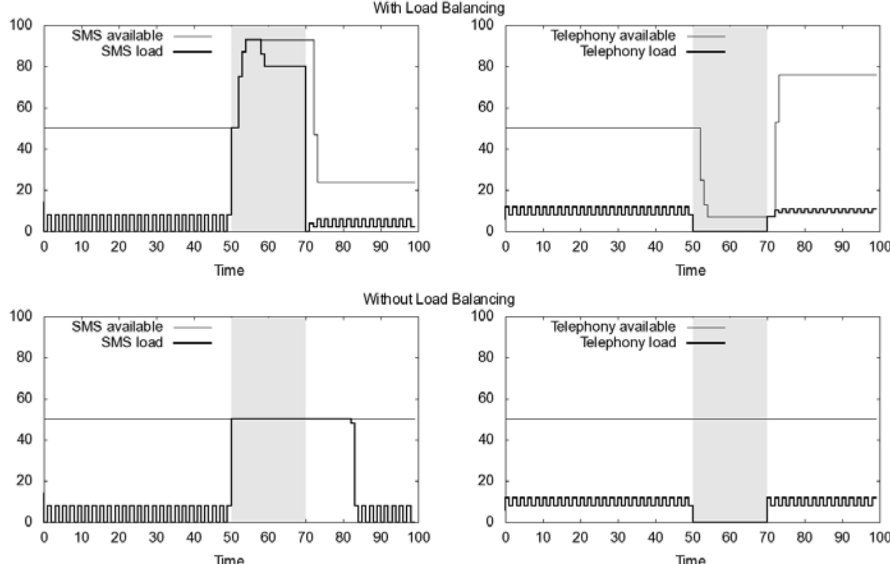


Fig. 6. Simulation of New Year's Eve behavior (SMS load spike between t=50 and t=70), with (top) and without resource balancing (bottom). The strategy of Fig. 5 distributes resources as needed between SMS component and telephony component.

in the unrestricted root component. Dynamic load balancing is captured by the `Balancer` class, an instance of which runs in parallel with the service in each component. This class implements a simple balancing strategy, transferring resources to its partner deployment component when receiving a `request` message (Line 12), and monitoring its own load and requesting assistance when needed (Line 9). Different, more involved or hierarchical, schemes for distributing resources among deployment components can be implemented similarly.

Figure 6 presents simulation results for this example scenario and for a scenario without load balancing, which shows that the available resources are sufficient for normal client behavior. In the load balancing scenario, the SMS service is overloaded between $t = 50$ and 53, at which time enough resources have been transfered from the telephony service to process the increased workload. After the load peak, the telephony service operates at capacity for one interval before receiving resources back from the SMS service. In the scenario without load balancing, the SMS service is overloaded during the whole load peak and another 12 time intervals while catching up with the backlog of delayed messages. Note that the functional part of the model was not changed between the two scenarios, and that more elaborate load balancing strategies can be added in similar ways.

5 Operational Semantics

The semantics of Creol is defined in rewriting logic (RL) [21], and Creol models can be analyzed using the rewrite tool Maude [9]. In a rewrite theory (Σ, E, L, R),

the signature Σ defines the ground terms, E defines equations between terms, L is a set of labels, and R a set of labeled rewrite rules. Rewrite rules apply to terms of given sorts, specified in (membership) equational logic (Σ, E). When modeling computational systems, different system components are typically modeled by terms of suitable sorts and the global state configuration is a set of these terms. RL extends algebraic specification techniques with transition rules which capture the dynamic behavior of a system. A *conditional rewrite rule* **crl** [*name*]: $t \longrightarrow t'$ **if** *cond* transforms an instance of the pattern t to evolve into the corresponding instance of the pattern t', where the condition *cond* is a conjunction of rewrites and equations that must hold for the main rule to apply (the *name* identifies the rule). When auxiliary functions are needed, these can be defined in equational logic, and thus evaluated in between the state transitions [21]. In a *conditional equation* **ceq** $t \ = t'$ **if** *cond* the condition must similarly hold for the equation to apply. Unconditional rewrite rules and equations are denoted by the keywords **rl** and **eq**, respectively. Given an initial configuration, Maude supports simulation and breadth-first search through reachable states and model checking of finite reachable states for desired properties. In this paper, Maude is used as an interpreter for Creol's operational semantics to simulate and test Creol models.

The States. Following Maude conventions runtime objects are represented by terms $\langle o : C | \ldots, \ \mathtt{Att}_i\mathtt{:}\ x_i, \ldots\rangle$, where o is the identifier, C the class, and the object contains a set of attributes such that $\mathtt{Att}_i$ is the name and x_i the current value of the i'th attribute. Variables are *slanted*, whereas constant parts of a term's syntax are in `typewriter` style. As before, $\overline{t}$ denotes a collection of terms t, either a list or a set depending on the context. Let `Emp` be the empty list and $\emptyset$ the empty set. In the rules below, all numbers are natural numbers (e.g., for time) except resources which are of sort `Resource`.

A state *configuration* is a set which consists of a global clock, deployment components, objects, classes, invocation messages, and futures. The associative and commutative union operator on configurations is denoted by whitespace and the empty configuration by `none`. The *entire* configuration lives inside curly brackets; thus, in the term $\{cn\}$ the variable cn captures the entire configuration. The global *clock* is a term $\langle t\colon \mathtt{Clock} \,|\, \mathtt{limit\colon}\ l \,\rangle$ where t is the current time and l the time limit considered in an execution. A *deployment component* is a term $\langle\, dc\colon \mathtt{Comp} \,|\, \mathtt{Free\colon}\ r, \mathtt{Limit\colon}\ max,\ \mathtt{Next\colon}\ next,\ \mathtt{Load\colon}\ \overline{m}\,\rangle$ where dc is the identifier of the component, r the (non-negative) number of available computing resources, max the maximum number of resources which can be consumed before time advances, *next* the maximum for the *next time interval*, and $\overline{m}$ the history of resource consumption over past time intervals.

An *object* is a term $\langle\, o\colon C \,|\, \mathtt{Att\colon}\ \overline{a},\ \mathtt{Pr\colon}\ \{\overline{l} \,|\, \overline{s}\},\ \mathtt{PrQ\colon}\ \overline{w},\ \mathtt{Lcnt\colon}\ f\,\rangle$ where o is the identifier and C the object's class, its state is given by the attribute mapping $\overline{a}$ (i.e., a single *binding* a binds a value to a declared variable), a *process* $\{\overline{l}\ |\overline{s}\}$ consists of a mapping $\overline{l}$ of local variable bindings and a list $\overline{s}$ of statements. The set $\overline{w}$ of (suspended) processes represents the process queue and the attribute

f is used to ensure that futures created by the object have unique identifiers (next(f) provides a new fresh value).

A *class* is a term $\langle C\colon \texttt{Class} \mid \texttt{Prm:}\ \overline{x}\texttt{, Att:}\ \overline{a}\texttt{, Mtds:}\ \overline{M}\texttt{, Ocnt:}\ g\,\rangle$ where C is the identifier, $\overline{x}$ the list of formal parameters, $\overline{a}$ maps declared attributes to default values, and $\overline{M}$ is the set of method definitions of the form $\langle m\colon \texttt{Mtd} \mid \texttt{Prm:}\ \overline{x}\texttt{, Att:}\ \overline{l}\texttt{, Code:}\ \overline{s}\,\rangle$. Here, m is the method name, $\overline{x}$ the formal parameter list, $\overline{l}$ the mapping of local variables to initial (default) values, and $\overline{s}$ a sequence of statements. The attribute g is used to create objects with unique identifiers.

An *invocation message* is a term $\texttt{invoc(}o\texttt{, }n\texttt{, }m\texttt{, }\overline{d}\texttt{)}$ where o is the callee, n the future to which the call's result is returned, m the method name, and $\overline{d}$ the call's actual parameter values. A *future* is a term $\langle n\colon \texttt{Fut} \mid \texttt{Done:}\ b\texttt{, Value:}\ d\,\rangle$ where n is the identifier, b a Boolean flag indicating whether the future's reply value has been received, and d the reply value.

Evaluating Expressions. Given a substitution σ, a time t and a configuration cn, we denote by $[\![e]\!]^{cn}_{\sigma,t}$ a confluent and terminating reduction system which reduces an expression e to a data value. Let $[\![\textbf{now}]\!]^{cn}_{\sigma,t} = t$, $[\![\textbf{mycomp}]\!]^{cn}_{\sigma,t} = \sigma[\textbf{mycomp}]$, the equations below define availability and resource load:

$$
\begin{array}{llll}
\textbf{ceq} & [\![\textbf{available}]\!]^{cn<dc\,:\text{Comp}|\texttt{Limit:}\ r>}_{\sigma,t} & = r & \textbf{if}\ dc = \sigma[\textbf{mycomp}] \\
\textbf{ceq} & [\![\textbf{load}(n)]\!]^{cn<dc\,:\text{Comp}|\texttt{Load:}\ \overline{m}>}_{\sigma,t} & = avg(\overline{m}, n) & \textbf{if}\ dc = \sigma[\textbf{mycomp}]
\end{array}
$$

$$
\begin{array}{lll}
\textbf{eq} & avg(emp, n) & = 0 \\
\textbf{eq} & avg(\overline{m} \circ m, n) & = \textbf{if}\ n > 0\ \textbf{then}\ avg(\overline{m}, n-1) + m/min(n, length(\overline{m})) \\
& & \qquad\qquad\quad \textbf{else}\ 0\ \textbf{fi}
\end{array}
$$

where $avg(\overline{m}, n)$ calculates the average number of used resources during the last n time intervals (or the average of $\overline{m}$ if its length is shorter than n). Let $[\![\text{x?}]\!]^{cn}_{\sigma,t} = \texttt{true}$ if $[\![\text{x}]\!]^{cn}_{\sigma,t} = n$ and there is a future $\langle n\colon \texttt{Fut} \mid \texttt{Done: true, Value:}\ d\,\rangle$ in cn (for some value d), otherwise $[\![\text{x?}]\!]^{cn}_{\sigma,t} = \texttt{false}$. The remaining cases of $[\![\text{e}]\!]^{cn}_{\sigma,t}$ are fairly straightforward, looking up values for declared variables in σ. Expressions are always reduced inside an object in a given state configuration. Thus, $\sigma = \overline{a} \circ \overline{l}$, the composition of the object state $\overline{a}$ and the local variable bindings $\overline{l}$, the time t is the current global time, and the configuration cn is the current global configuration (ignoring the object itself). This ensures that **now**, **mycomp**, **available**, and **load(**n**)**, as well as reply guards and declared variables, are evaluated correctly in the state of the program.

Transitions. Rewrite rules transform state configurations into new configurations, and are given in Fig. 7. In the presentation of a rule, we follow the convention of Full Maude [9] and hide attributes in runtime objects unless they are needed for that specific rule. Rule *skip* consumes a **skip** in the active process and a resource in its deployment component. Rule *assign* evaluates an expression e and assigns the value to a variable x in the local state $\overline{l}$ or in the attributes $\overline{a}$, as appropriate, consuming a resource in its deployment component. (The rules for **if** and **while** statements are omitted from the presentation.)

$$\mathbf{crl}\ [\mathit{skip}]:\ \langle\, o : C \mid \texttt{Pr: } \{\overline{l} \mid \texttt{skip;}\overline{s}\}\,\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r\,\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{Pr: } \{\overline{l} \mid \overline{s}\}\,\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r-1\,\rangle\ \mathbf{if}\ dc = \overline{a}[\texttt{mycomp}]\ .$$

$$\mathbf{crl}\ [\mathit{assign}]:\ \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid x\texttt{:=}e\texttt{;}\overline{s}\}\,\rangle\ \langle\, t\texttt{: Clock} \mid\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r\,\rangle$$
$$\longrightarrow \mathbf{if}\ x \in \texttt{dom}(\overline{l})\ \mathbf{then}\ \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l}[x \mapsto [\![e]\!]^{none}_{(\overline{a}\circ\overline{l}),t}] \mid \overline{s}\}\,\rangle$$
$$\mathbf{else}\ \langle\, o : C \mid \texttt{Att: } \overline{a}[x \mapsto [\![e]\!]^{none}_{(\overline{a}\circ\overline{l}),t}]\texttt{, Pr: } \{\overline{l} \mid \overline{s}\}\,\rangle\ \mathbf{fi}$$
$$\langle\, t\texttt{: Clock} \mid\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r-1\,\rangle\ \mathbf{if}\ dc = \overline{a}[\texttt{mycomp}]\,.$$

$$\mathbf{crl}\ [\mathit{return}]:\ \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid \texttt{return(}e\texttt{);}\overline{s}\}\,\rangle\ \langle\, t\texttt{: Clock} \mid\rangle$$
$$\langle\, n\texttt{: Fut} \mid \texttt{Done: false, Value: } \bot\,\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r\,\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid \overline{s}\}\,\rangle\ \langle\, n\texttt{: Fut} \mid \texttt{Done: true, Value: } [\![e]\!]^{none}_{(\overline{a}\circ\overline{l}),t}\,\rangle$$
$$\langle\, t\texttt{: Clock} \mid\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r-1\,\rangle\ \mathbf{if}\ n = \overline{l}(\texttt{destiny})\ \wedge\ dc = \overline{a}[\texttt{mycomp}]\ .$$

$$\mathbf{rl}\ [\mathit{release}]: \langle\, o : C \mid \texttt{Pr: } \{\overline{l} \mid \texttt{release;}\overline{s}\}\texttt{, PrQ: } \overline{w}\,\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{Pr: idle, PrQ: } \texttt{enqueue}(\{\overline{l} \mid \overline{s}\}, \overline{w})\,\rangle\ .$$

$$\mathbf{crl}\ [\mathit{await1}]:\ \{\langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid \texttt{await } e\texttt{;}\overline{s}\}\,\rangle\ \mathit{cn}\ \langle\, t\texttt{: Clock} \mid\rangle\,\}$$
$$\longrightarrow \{\langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid \overline{s}\}\,\rangle\ \mathit{cn}\ \langle\, t\texttt{: Clock} \mid\rangle\,\}\ \mathbf{if}\ [\![e]\!]^{cn}_{(\overline{a}\circ\overline{l}),t}\ .$$

$$\mathbf{crl}\ [\mathit{await2}]:\ \{\langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid \texttt{await } e\texttt{;}\overline{s}\}\,\rangle\ \mathit{cn}\ \langle\, t\texttt{: Clock} \mid\rangle\,\}$$
$$\longrightarrow \{\langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid \texttt{release;await } e\texttt{;}\overline{s}\}\,\rangle\ \mathit{cn}\ \langle\, t\texttt{: Clock} \mid\rangle\,\}\ \mathbf{if}\ \neg[\![e]\!]^{cn}_{(\overline{a}\circ\overline{l}),t}\ .$$

$$\mathbf{crl}\ [\mathit{activate}]:\ \{\langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: idle, PrQ: } \overline{w} \cup \{\{\overline{l} \mid \overline{s}\}\}\,\rangle\ \mathit{cn}\ \langle\, t\texttt{: Clock} \mid\rangle\,\}$$
$$\longrightarrow \{\langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } p\texttt{, PrQ: } \texttt{dequeue}(\overline{w}, p)\rangle\!\rangle\ \mathit{cn}\ \langle\, t\texttt{: Clock} \mid\rangle\}$$
$$\mathbf{if}\ p = \texttt{select}(\overline{w}, \overline{a}, \mathit{cn}, t)\ .$$

$$\mathbf{crl}\ [\mathit{async\text{-}call}]:\ \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid x\texttt{:=}e\texttt{!}m\texttt{(}\overline{e}\texttt{);}\overline{s}\}\texttt{, Lcnt: } f\,\rangle\ \langle\, t\texttt{: Clock} \mid\rangle$$
$$\langle\, dc\texttt{: Comp} \mid \texttt{Free: } r\,\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l}[x \mapsto n] \mid \overline{s}\}\texttt{, Lcnt: } \texttt{next}(f)\,\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r-1\,\rangle$$
$$\texttt{invoc(}[\![e]\!]^{none}_{(\overline{a}\circ\overline{l}),t}, n, m, [\![\overline{e}]\!]^{none}_{(\overline{a}\circ\overline{l}),t}\texttt{)}\ \langle\, n\texttt{: Fut} \mid \texttt{Done: false, Value: } \bot\,\rangle\ \langle\, t\texttt{: Clock} \mid\rangle$$
$$\mathbf{if}\ n = \texttt{label}(o, f) \wedge o \neq [\![e]\!]^{none}_{(\overline{a}\circ\overline{l}),t}\ \wedge\ dc = \overline{a}[\texttt{mycomp}]\ .$$

$$\mathbf{rl}\ [\mathit{bind\text{-}method}]:\ \texttt{invoc(}o, n, m, \overline{d}\texttt{)}\ \langle\, o : C \mid \texttt{PrQ: } \overline{w}\,\rangle$$
$$\langle\, C\texttt{: Class} \mid \texttt{Mtds: (}\overline{M} \cup \{\langle\, m\texttt{: Mtd} \mid \texttt{Prm: } \overline{x}\texttt{, Att: } \overline{l}\texttt{, Code: } \overline{s}\,\rangle\,\}\rangle\!\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{PrQ: } \overline{w} \cup \{\{\overline{l}[\texttt{destiny} \mapsto n, \overline{x} \mapsto \overline{d}] \mid \overline{s}\}\}\,\rangle$$
$$\langle\, C\texttt{: Class} \mid \texttt{Mtds: (}\overline{M} \cup \{\langle\, m\texttt{: Mtd} \mid \texttt{Prm: } \overline{x}\texttt{, Att: } \overline{l}\texttt{, Code: } \overline{s}\,\rangle\,\}\rangle\!\rangle\ .$$

$$\mathbf{crl}\ [\mathit{receive\text{-}comp}]:\ \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid x\texttt{:=}e\texttt{.get;}\overline{s}\}\,\rangle$$
$$\langle\, n\texttt{: Fut} \mid \texttt{Done: true, Value: } d\,\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r\,\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid x\texttt{:=}d\texttt{; } \overline{s}\}\,\rangle\ \langle\, n\texttt{: Fut} \mid \texttt{Done: true, Value: } d\,\rangle$$
$$\langle\, dc\texttt{: Comp} \mid \texttt{Free: } r-1\,\rangle\ \mathbf{if}\ n = [\![e]\!]^{none}_{(\overline{a}\circ\overline{l}),t} \wedge\ \texttt{dc} = \overline{a}[\texttt{mycomp}]\ .$$

$$\mathbf{crl}\ [\mathit{object\text{-}creation}]:\ \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid x\texttt{:=new } B\texttt{(}\overline{e}\texttt{);}\overline{s}\}\,\rangle\ \langle\, t\texttt{: Clock} \mid\rangle$$
$$\langle\, dc\texttt{: Comp} \mid \texttt{Free: } r\,\rangle\ \langle\, B\texttt{: Class} \mid \texttt{Prm: } \overline{x}\texttt{, Att: } \overline{a}_1\texttt{,}$$
$$\texttt{Mtds: } \overline{M} \cup \{\langle\, \texttt{init: Mtd} \mid \texttt{Prm: Emp, Att: } \emptyset\texttt{, Code: } \overline{s}_1\,\rangle\,\}\texttt{, Ocnt: } g\,\rangle$$
$$\longrightarrow \langle\, o : C \mid \texttt{Att: } \overline{a}\texttt{, Pr: } \{\overline{l} \mid x\texttt{:=newId(}B, g\texttt{);}\overline{s}\}\,\rangle\ \langle\, B\texttt{: Class} \mid \texttt{Prm: } \overline{x}\texttt{, Att: } \overline{a}_1\texttt{,}$$
$$\texttt{Mtds: } \overline{M} \cup \{\langle\, \texttt{init: Mtd} \mid \texttt{Prm: Emp, Att: } \emptyset\texttt{, Code: } \overline{s}_1\,\rangle\,\}\texttt{, Ocnt: } \texttt{next}(g)\,\rangle$$
$$\langle\, \texttt{newId}(B, g) : B \mid \texttt{Att: } \overline{a}_1[\texttt{mycomp} \mapsto dc\texttt{, this} \mapsto \texttt{newId(}B, g\texttt{)}, \overline{x} \mapsto [\![\overline{e}]\!]^{none}_{\overline{a}\circ\overline{l},t}]\texttt{,}$$
$$\texttt{Pr: } \{\emptyset \mid \overline{s}_1\}\texttt{, PrQ: } \emptyset\texttt{, Lcnt: } 0\,\rangle\ \langle\, t\texttt{: Clock} \mid\rangle\ \langle\, dc\texttt{: Comp} \mid \texttt{Free: } r-1\,\rangle$$
$$\mathbf{if}\ dc = \overline{a}[\texttt{mycomp}]\ .$$

Fig. 7. A timed rewriting logic semantics for Creol. In the rewrite rules, the variable r ranges over non-zero natural numbers to ensure that resource values are non-negative. The rules for the **`if`** and **`while`** statements are standard and not shown in this figure.

Process suspension and activation. Three operations are used to manipulate the process queue $\overline{w}$: enqueue$(p, \overline{w})$ adds a process p to $\overline{w}$, select$(\overline{w}, \overline{a}, cn, t)$ selects a process from $\overline{w}$ (if $\overline{w}$ is empty or no process is *ready* [17], this is the idle process), and dequeue$(\overline{w}, p)$ removes the process p from $\overline{w}$. The actual definitions of enqueue and select are left undefined; different definitions correspond to different scheduling policies for processes and can be used to locally express, e.g., priority or fairness. Rule *release* suspends the active process to the process queue. We denote by `idle` the idle process. Rule *await1* consumes the await statement if the guard evaluates to true in the current state, rule *await2* adds a `release` statement in order to suspend the process if the guard evaluates to false. Rule *activate* selects a process from the process queue for execution if this process is *ready* to execute, i.e., if it would not directly be resuspended or block the processor [17].

Communication and object creation. Rule *async-call* sends an invocation message to a callee with the actual method parameters and the identity of a future in which to place the method's return value. The caller creates the future associated with the call, with a unique identity `label`(o, f) constructed from the caller's own identity o and the local attribute f. The future's `Done` attribute is initially `false` and the return value is undefined (i.e., $\perp$). This operation consumes a resource. Rule *bind-method* transforms a method invocation into a corresponding process, placed in the process queue of the callee. The reserved local variable `destiny` stores the identity of the call's future. Rule *return* puts the return value into the future associated with the call (the `destiny`-variable refers to the appropriate future) and sets the future's `done` attribute to true. This operation consumes a resource. Rule *receive-comp* dereferences the future variable n in the case where the future's `Done` attribute is `true`. Note that if this attribute is `false` the reduction in this object is *blocked*. This operation consumes a resource. Finally, *object-creation* creates a new object with a unique identifier `newId`(B, g) constructed from the class identifier B and the local attribute g. The object's state is generated from default values for state attributes, extended with the actual values for `this` and the class parameters. The `init` method is loaded (we assume that this method reduces to `skip` if unspecified and that it asynchronously calls `run` if the latter is specified). This operation consumes a resource. Note that the new object inherits the deployment component of its creator. The rule for object creation in a named deployment component differs from *object-creation* only on this point, and is not presented.

Advancing time. We define a *run-to-completion* semantics for execution with the resource bounds of deployment components: objects must execute when possible if resources are available. To capture timed concurrent execution with an interleaving semantics, time cannot advance freely but is restricted as follows:

- For simplicity, we assume that invocation messages do not take time. Therefore, time may *not* advance while a message is on its way.
- If a deployment component has run out of resources, none of its objects may proceed, and time can advance.

```
eq canAdv(cn′,t) = true .                    //cn′ contains no objects or messages
eq canAdv(msg cn,t) = false .                        //messages are instantaneous
eq canAdv(⟨ o : C | ⟩ ⟨ dc: Comp | Free: 0 ⟩ cn,t)              //no more resources
   = canAdv(⟨ dc: Comp | Free: 0 ⟩ cn,t).
eq canAdv(⟨ o : C | Pr: {l̄ | n.get;s̄)}⟩        //o is blocked, value not available
   ⟨ n: Fut | Done: false ⟩ cn,t) = canAdv(⟨ n: Fut | Done: false ⟩ cn,t).
ceq canAdv(⟨ o : C | Att: ā,Pr: idle,PrQ: w̄ ⟩ cn,t)          //no ready processes
   = canAdv(cn,t) if select(w̄, ā, cn, t) = idle.
eq canAdv(⟨ o : C | ⟩ cn,t) = false [owise] .

eq Adv(⟨ dc: Comp | Free: r,Limit: max, Next: next, Load: m̄ ⟩ cn) =
⟨ dc: Comp | Free: next,Limit: next, Next: next, Load: m̄ ∘ max − r ⟩ Adv(cn).
eq Adv(cn) = cn [owise] .

crl [progress]: {cn ⟨ t: Clock | limit: l ⟩} ⟶ {Adv(cn) ⟨ t + 1: Clock | limit: l ⟩}
if canAdv(cn,t) ∧ t < l .

crl [resource-transfer]: ⟨ o : C | Att: ā, Pr: {l̄ | transfer(e,e1);s̄} ⟩
⟨ dc1: Comp | Next: nl ⟩ ⟨ dc2: Comp | Next: nl1 ⟩ cn ⟨ t: Clock |⟩
⟶ ⟨ o : C | Att: ā, Pr: {l̄ | s̄} ⟩ cn ⟨ t: Clock |⟩ ⟨ dc1: Comp | Next: nl − d ⟩
⟨ dc2: Comp | Next: nl1 + d ⟩ if [[e]]^cn_(ā∘l̄,t) = dc2 ∧ [[e1]]^cn_(ā∘l̄,t) = d ∧ nl ≥ d ∧ dc1 = ā[mycomp] .
```

Fig. 8. Advancing time and transferring resources. The variable *msg* denotes a message, r a non-zero natural number, and *cn'* a message- and object-free configuration.

- If a deployment component has remaining resources and one of the component's objects o may execute, time may *not* advance. There are three cases:
  1. The active process in o is blocked, but the value has become available.
  2. The active process in o is idle, but a suspended process can be activated.
  3. The active process in o is not blocked.

A predicate canAdv, defined recursively over configurations (see Fig. 8), formalizes these restrictions on time advance in an interleaving semantics for timed concurrent execution. Time may not advance if some object can execute, expressed by the **owise** equation for canAdv. (The keyword **owise** in Maude expresses that an equation is chosen only when no other equation applies.) Finally time may advance if no object can execute and there are no messages, which is captured by the first equation for canAdv. Once time advances, the global clock is updated and the deployment components get their resources refreshed for the next time interval. This is done by an auxiliary function Adv defined in Fig. 8, which updates a configuration by resetting the free resources of each deployment component to the limit specified by *next* and extending the load history of the components. (Here, $\overline{m} \circ m$ appends m to the sequence $\overline{m}$.)

The advancement of time is captured by the rewrite rule *progress* in Fig. 8. Observe that for simplicity time advances with a single unit. It would be straightforward to allow larger increments. In order to ensure termination of model execution, a *limit* has been added to the global clock and we only consider execution sequences up to this limit in time.

6 Related Work

Concurrent objects and Actors, in which software units with encapsulated processors communicate asynchronously, increasingly attract attention due to an

intuitive and compositional concurrency model [2, 4, 6, 26, 13, 10, 3]. Creol proposes cooperative scheduling between asynchronously called methods [17], which allows active and reactive behavior to be combined within objects as well as compositional verification of partial correctness properties [10,3]. This model of cooperative scheduling has recently been generalized to concurrent object groups in Java [24]. This paper further generalizes concurrent object groups to resource-constrained deployment components, where group activity per time interval is parametric in concurrent resources, using a time model which simplifies previous work [18]. The approach abstractly models the effect of deploying concurrent object groups on deployment components which vary in processing capacity.

Techniques and methodologies for predictions or analysis of non-functional properties are based on either *measurement* or *modeling*. Measurement-based approaches apply to existing implementations, using dedicated profiling or tracing tools like, e.g., JMeter or LoadRunner. Model-based approaches allow abstraction from specific system intricacies, but depend on parameters provided by domain experts [11]. A survey of model-based performance analysis techniques is given in [5]. Formal approaches using process algebra, Petri Nets, game theory, and timed automata (e.g., [7,8,12,15,19,20]) have been applied in the embedded software domain, but also to the schedulability of tasks in concurrent objects [16]. That work complements ours as it does not consider resource restrictions on the concurrency model, but associates deadlines with method calls.

Work on modeling object-oriented systems with resource constraints is more scarce. Using the UML SPT profile for schedulability, performance and time, Petriu and Woodside [22] informally define the Core Scenario Model (CSM) to solve questions that arise in performance model building. CSM has a notion of resource context, which reflects the set of resources used by an operation. CSM aims to bridge the gap between UML specifications and techniques to generate performance models [5]. UML models with stochastic annotations for performance prediction have been proposed for components [14]. Closer to our work is M. Verhoef's extension of VDM++ for simulation of embedded real-time systems [25], in which architectures are explicitly modeled using CPUs and buses, and resources are statically bound to the CPUs. Our work extends these ideas with dynamic load balancing strategies expressed in the modeling language and running in parallel with the behavioral parts of the model.

7 Conclusions and Future Work

We present a modeling framework which formalizes a high-level understanding of deployment concerns, reflecting the execution capabilities of underlying architectures. This framework is based on an abstract notion of execution resource, such that each component has an associated amount of available resources which can be used within a time interval. The framework is given as an extension of the object-oriented language Creol, allowing the dynamic creation of deployment components and the dynamic reallocation of resources, such that redistribution strategies can be expressed in terms of the load and the available resources of

components. Resources and deployment components have been naturally integrated as first-class values at the abstraction level of the modeling language, including constructs to transfer resources, create deployment components and place new objects in given deployment components, as well as to check the current load of a component and its available resources. The extended language has been formalized by a timed operational semantics in rewriting logic. Rewriting logic semantics are directly executable in Maude, which allows the tool-supported simulation and analysis of models directly based on the operational semantics.

As shown by an example, the approach is compositional in the sense that the software controlling allocation and reallocation of resources can (but need not) be completely separated from the rest of the code. Classes express particular reallocation strategies, and one strategy object is created in each component that should be controlled by that strategy. It is easy to replace a strategy by another, to reuse strategies, and to apply different strategies to different components. This flexibility is valuable for software development with high needs for deployment configurability; for example in software product lines, variability in resources and reallocation strategies allow products to be deployed on different architectures while maintaining, e.g., response time requirements.

The proposed notions of resource and time stem from the need for abstract models which do not assume a fixed deployment scenario, yet support tool-based formal analysis and model exploration. However, our approach may be extended with more fine-grained notions of resources and resource consumption; e.g., using resource profiles for specific deployment scenarios. In future work, we plan to develop case studies using reallocation strategies based on gossiping, peer-to-peer, and hierarchical structures, as well as object migration. Furthermore, it is interesting to combine the simulation-based approach with concrete values in Maude with symbolic execution techniques for resource consumption and reallocation.

References

1. Ábrahám-Mumm, E., de Boer, F.S., de Roever, W.-P., Steffen, M.: Verification for Java's reentrant multithreading concept. In: Nielsen, M., Engberg, U. (eds.) FOSSACS 2002. LNCS, vol. 2303, pp. 5–20. Springer, Heidelberg (2002)
2. Agha, G.A.: ACTORS: A Model of Concurrent Computations in Distributed Systems. The MIT Press, Cambridge (1986)
3. Ahrendt, W., Dylla, M.: A verification system for distributed objects with asynchronous method calls. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 387–406. Springer, Heidelberg (2009)
4. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
5. Balsamo, S., Marco, A.D., Inverardi, P., Simeoni, M.: Model-based performance prediction in software development: A survey. IEEE Trans. on Software Engineering 30(5), 295–310 (2004)
6. Caromel, D., Henrio, L.: A Theory of Distributed Object. Springer, Heidelberg (2005)
7. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)

8. Chen, X., Hsieh, H., Balarin, F.: Verification approach of metropolis design framework for embedded systems. Intl. J. Parallel Programming 34(1), 3–27 (2006)
9. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science 285, 187–243 (2002)
10. de Boer, F.S., Clarke, D., Johnsen, E.B.: A complete guide to the future. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 316–330. Springer, Heidelberg (2007)
11. Epifani, I., Ghezzi, C., Mirandola, R., Tamburrelli, G.: Model evolution by run-time parameter adaptation. In: Proc. ICSE 2009, pp. 111–121. IEEE, Los Alamitos (2009)
12. Fersman, E., Krcál, P., Pettersson, P., Yi, W.: Task automata: Schedulability, decidability and undecidability. Inf. and Comp. 205(8), 1149–1172 (2007)
13. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science 410(2-3), 202–220 (2009)
14. Happe, J., Koziolek, H., Reussner, R.: Parametric performance contracts for software components with concurrent behaviour. In: Proc. 3rd Intl. Workshop on Formal Aspects of Component Software (FACS 2006). ENTCS, vol. 182, pp. 91–106 (2007)
15. Hennessy, M., Riely, J.: Information flow vs. resource access in the asynchronous pi-calculus. ACM Trans. on Prog. Languages and Systems 24(5), 566–591 (2002)
16. Jaghoori, M.M., de Boer, F.S., Chothia, T., Sirjani, M.: Schedulability of asynchronous real-time concurrent objects. Journal of Logic and Algebraic Programming 78(5), 402–416 (2009)
17. Johnsen, E.B., Owe, O.: An asynchronous communication model for distributed concurrent objects. Software and Systems Modeling 6(1), 35–58 (2007)
18. Johnsen, E.B., Owe, O., Bjørk, J., Kyas, M.: An object-oriented component model for heterogeneous nets. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 257–279. Springer, Heidelberg (2008)
19. Katelman, M., Meseguer, J., Hou, J.C.: Redesign of the lmst wireless sensor protocol through formal modeling and statistical model checking. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 150–169. Springer, Heidelberg (2008)
20. Katoen, J.-P., Baier, C., Latella, D.: Metric semantics for true concurrent real time. Theoretical Computer Science 254(1-2), 501–542 (2001)
21. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96, 73–155 (1992)
22. Petriu, D.B., Woodside, C.M.: An intermediate metamodel with scenarios and resources for generating performance models from UML designs. Software and System Modeling 6(2), 163–184 (2007)
23. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
24. Schäfer, J., Poetzsch-Heffter, A.: JCoBox: Generalizing active objects to concurrent components. In: D'Hondt, T. (ed.) ECOOP 2010 – Object-Oriented Programming. LNCS, vol. 6183, pp. 275–299. Springer, Heidelberg (2010)
25. Verhoef, M., Larsen, P.G., Hooman, J.: Modeling and validating distributed embedded real-time systems with VDM++. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 147–162. Springer, Heidelberg (2006)
26. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proc. OOPSLA 2005, pp. 439–453. ACM, New York (2005)
27. Yacoub, S.M.: Performance analysis of component-based applications. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 299–315. Springer, Heidelberg (2002)

A Pattern System to Support Refining Informal Ideas into Formal Expressions*

Xi Wang[1], Shaoying Liu[1], and Huaikou Miao[2]

[1] Department of Computer Science, Hosei University, Japan
[2] Department of Computer Science, Shanghai University, China

Abstract. Refining informal ideas into appropriate formal expressions is an essential and skillful activity in writing pre-post style formal specifications. This activity usually involves decisions to be made by the writer and can be error-prone. Experience shows that this activity is also a challenge to many practitioners, and a big hurdle for introducing formal specification techniques into industry. This paper describes a *pattern system* approach to deal with this problem. The pattern system is composed of a set of inter-related patterns, and each pattern provides a framework for constructing certain kind of formal expression with some common properties. Unlike the way conventional design patterns are used, our pattern system is expected to support a systematic and automated formalization of informal ideas, with the characteristic that the writer only needs to work on the informal level while an appropriate formal expression will be efficiently derived. We focus on discussions of the issues such as pattern definition, pattern classification, the structure of pattern system, and mechanism to use the pattern system.

1 Introduction

In spite of the successful stories reported in the latest survey on industrial use of formal methods by Woodcock *et al.* [14], applications of formal methods to real projects in industrial are still rare [10]. Experience shows that industrial practitioners are interested in using formal methods to solve their problems occurring in exercising conventional software engineering techniques, but only a few of them with courage actually take actions [11,6]; most of them turn away from the techniques after they learn or try them for the first time, because they face insurmountable challenges and complexity, even they may understand the potential benefits of successful application of formal methods.

One of the major challenges is to formalize informal requirements into pre-post style formal specifications. It is often the case that the writer of a formal specification, who can be an analyst or designer, understands what he or she wants to say, but does not know exactly what formal expression can be used to properly

* This work is supported in part by NII Collaborative Research Program. Shaoying Liu is also supported by the NSFC Grant (No. 60910004) and 973 Program of China Grant (No. 2010CB328102). This work is also supported by Science and Technology Commission of Shanghai Municipality under Grant No. 10510704900 and Shanghai Leading Academic Discipline Project(Project Number: J50103).

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 662–677, 2010.
© Springer-Verlag Berlin Heidelberg 2010

express his or her informal idea. This difficulty may attribute to the writer lacking sufficient ability or experience in using formal notation, but because of this hurdle many beginners would stop continuing to learn the techniques in training courses and to use them in real projects [8].

In fact, refining informal ideas into formal expressions is an essential and skillful activity in constructing pre-post style formal specifications, and this activity usually involves decisions to be made by the specification writer in order to clarify ambiguities. An "idea" describes a fact or an intention; it may be expressed informally or formally. For example, "John Smith belongs to Hosei University" is an informal expression of an idea. How to refine it into an appropriate formal expression will depend on what we mean by "John Smith" and "Hosei University". If "John Smith" is treated as a person and "Hosei University" as a set of persons, a membership expression (e.g., John_Smith in set Hosei_University in VDM [13]) can be an appropriate formal expression. But if the data structure of "Hosei University" is declared as a set of faculties and each faculty is a composite type of several fields, for example, "teachers", "students", and "administrators", and each field is declared as a set of persons, the formal expression of the above idea will be more complex. Experience shows that this refining process can be error-prone and challenging to many practitioners. Without effectively attacking this problem, introducing formal specification techniques to ordinary practitioners in industry would be extremely difficult.

In this paper, we put forward a *pattern system* approach to tackle this problem. The pattern system is composed of a set of inter-related patterns, each providing a framework for constructing certain kind of formal expression with some common properties. The concept of pattern was initially proposed by Alexander who is an expert in the area of architecture rather than software engineering [1]. He designed each pattern to handle certain reoccurring problems with concrete solutions in certain context. The idea was later adopted by software engineers in software design[3] and many other fields including formal specification. Although patterns offer great convenience to practitioners, they are often complained of hard to be applied, because they are so general that users have to read them carefully and make a good understanding of them to select appropriate ones and apply them to specific problems. This situation may be exacerbated when the number and varieties of patterns become greater [2]. Unlike the way conventional patterns are used, our pattern system is expected to support a systematic and automated formalization of informal ideas with the characteristic that the writer only needs to work on the informal level while an appropriate formal expression will be efficiently derived. This will allow the writer to concentrate only on the ideas, while manipulation of formal notation to form the most appropriate formal expressions can leave to the machine. Consequently, the formalization process would become easier and mistakes would be reduced significantly.

Our contribution in this paper concentrates only on the theoretical discussion of the pattern system, including patter definition, pattern classification, the structure of pattern system, and mechanism to use the pattern system. The purpose is to set up a foundation for developing an automated tool support

in the future. The essential idea of our patter system is language independent; it can be applied to any model-based formal specification languages that use the notion of pre- and post-conditions for operation specifications. Since our research is required to directly support the Structured Object-Oriented Formal Language (SOFL) [7], an extension of VDM-SL [13] by integrating the essential concept of pre- and post-conditions into the conventional requirements and design techniques (e.g., data flow diagrams, object-oriented methods) for industrial application, we use SOFL as the "working language" for discussions in this paper.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 discusses the pattern system in two aspects: static structure of the system and mechanism for using the system. Section 4 gives an example to illustrate the proposed pattern system. Finally, in Section 5 we conclude the paper and point out future research directions.

2 Related Work

In contrast to design patterns [3], *formal specification pattern* is a relatively new area, and only a few researches have been reported in the literature. Stepney *et al.* proposed patterns for notation Z and classified them into six types for solving different kinds of problems [12]. Those patterns mainly concentrate on the structure of various kinds of Z schemas. Lars Grunske presented a specification pattern system of common probabilistic properties for probabilistic verification [4]. Ding *et al.* gave an approach for specification construction through property-preserving refinement patterns [5]. Matthew *et al.* applied patterns to presentation, codification and reuse of property specification in a range of common formalisms [9].

In comparison with the above related work, our pattern system, presented in this paper, concentrates only on the support of refining informal ideas into formal expressions in writing pre- and post-conditions, and aims to take a different approach to support the use of patterns. In our approach, the pattern system is applied by algorithms on the computer to provide effective guidance to the writer and to help him or her, in an interactive manner, gradually refine informal ideas into formal expressions. Thus, the writer neither needs to study the patterns defined in the pattern system, nor needs to select a specific one from a mass of patterns for use; what the writer needs to do is only to answer questions raised by the potential tool built based on the pattern system. This interactive process helps the writer clarify ambiguities and possibly discover new ideas.

3 Pattern System

This section presents the pattern system in four aspects: pattern definition, pattern classification, structure of the pattern system, and the mechanism for applying the system. It should be noted that types and variables play important roles in formal specifications, but since this paper focuses on the construction

process of formal descriptions, we assume that all the types and variables are already defined properly.

3.1 Pattern Definition

Our pattern system is composed of single patterns, each of which is intended to support refinement of a kind of informal idea. We define an individual pattern as the following structure:

name	The name of the pattern
explanation	Situations or functions that the pattern can describe
constituents	A set of elements necessary for applying the pattern
syntax	Grammatical rule for writing expressions using the pattern
solution	Method for the generation of the formal expression

Serving as a unique identity, *name* is used to invoke the corresponding pattern; *explanation* provides a brief introduction to the pattern in natural language.

Item *constituents* contains the elements that have to be specified when applying the pattern. For example, in the pattern *belongTo*, as shown in Fig 1, the *constituents* owns two elements *element* and *container* which stand for a member and its belonging collection respectively. To apply such pattern, these two elements are required to be specified, that is, to designate a value to each of them through defined variables. But elements would become too complex to be presented in this way, we therefore classify them from three perspectives: concrete or abstract, value or choice, static or dynamic.

name: belongTo
explanation: This pattern can be used to describe that an element is part of certain object
constituents: element, container
syntax: belongTo(element,container)
solution:

(dataType(element), dataType(container))	→	formal expression
(T, **set of** T)	→	"element **inset** container"
(**set of** T, **set of** T)	→	"**subset**(element, container)"
(T, **seq of** T)	→	"element **inset** **elems**(container)"
(**seq of** T, **set of** T)	→	"**exists**[i, j: **nat0**] | element = container(i, j)"
(T, **map** T **to** T')	→	"element **inset** **dom**(container)"
	…	

Fig. 1. Parts of the pattern "belongTo"

Concrete element must be directly specified while abstract element, which is a combination of children elements including abstract elements and concrete elements, is specified by specifying all its children elements. Concrete element is further divided into two kinds because of their different meanings: value element and choice element. Value element requires to be designated with values such as the above *element* and *container*. By contrast, choice element denotes a choice item that is specified by selecting it or not. To distinguish these two

elements, we set "&" as the prefix of choice element. For abstract element, several symbols are introduced to formally present the relation between children elements. Expression "$e : e_1 \wedge e_2 \wedge ... \wedge e_n$" indicates that $e_1, e_2, ..., e_n$ are children elements that must be specified for the abstract element e to be specified. Suppose each e_i is given a value v_i, the value of e can then be presented as a tuple: $(v_1, v_2, ..., v_n)$. Expression "$e : e_1 \vee e_2 \vee ... \vee e_n$" denotes that at least one children element e_i must be specified and the value of e will appear as a tuple $(item_1, item_2, ..., item_m)$ where $m \leq n$ and each $item_i$ is the value of a value element or a selected choice element. The use of "$\|$" is similar to "$\vee$" with the difference that only one children element connected by "$\|$" needs to be given a value or selected and the value of e is an item $item_i$ instead of a tuple.

Elements in the *constituents* are pre-defined, however, they are not always asked for clarification at a time, because children elements of some abstract elements, as well as the contents of some basic elements, vary with different values of other elements. These dynamically determined elements are called dynamic elements and those which stay the same under all conditions are static elements. Instead of simply listing these two kinds of elements in an ad hoc manner, we define them by pairs and mappings.

Let's take the pattern *alter* as an example, part of which is given in Fig 2 where *dataType*() indicates the data type of a given variable and *constraints*() denotes certain feature of a given object. It contains three elements in *constituents*, including *object*, $specifier$ and *operation* where *object* stands for the variable to be altered; $specifier$ depicts the specific data items to be altered within the variable; *operation* demonstrates the new value, denoted as $newValue$, for replacing the original one, and the way to obtain it denoted as *operationType*. There are three pairs ($object, specifier$), ($specifier, operation$) and ($operationType, newValue$) with detailed rules given right below each of them. The first pair means that the definition of $specifier$ depends on the value the user designates to *object*. For instance, a map-type *object* will lead to a $specifier$ that identifies the specific maplets to be altered by addressing constraints on domain, range or relations between domain and range, and specifies the target parts to be altered in these specific maplets by choosing from their domain, range or both; the second pair reveals that for each item in the value of $specifier$, there must be a corresponding item in *operation*; the last pair illustrates that the value of *operationType* determines whether $newValue$ needs to be given a value. Thus, *object* and *operationType* are static elements while $specifier$, *operation* and $newValue$ are dynamic ones.

The fourth item *syntax* in a pattern contains a standard expression consisting of the pattern name and the elements of *constituents* in order. It is designed for being recognizable to the potential tool in the case that all the elements in *constituents* are available so that the pattern can be applied automatically.

The last item *solution* indicates the rules mapping from a kind of feature of given elements to a result that is either a suggested formal fragment or an

name: alter
explanation: common modification on a system variable or certain parts of a system variable
constituents: object, specifier, operation
(object, specifier)
dataType(object) = map →
specifier : (constraints$_1$(dom) ∨ constraints$_1$(rng) ∨ constraints$_1$(dom, rng)) ∧ (&dom ∨ &rng)
∨ ⋯
∨ (constraints$_n$(dom) ∨ constraints$_n$(rng) ∨ constraints$_n$(dom, rng)) ∧ (&dom ∨ &rng)

dataType(object) = composed of field$_1$: T$_1$ … field$_n$: T$_n$ → specifier : &field$_1$ ∨ … ∨ &field$_n$
…
(specifier, operation)
specifier = (item$_1$, …, item$_i$,…) →
operation: (operationType$_1$ ∧ newValue$_1$) ∧ … ∧ (operationType$_i$ ∧ newValue$_i$) ∧ …
operationType : &customize ‖ &update ‖ &retrieve ‖ &recreate
(operationType, newValue)
operationType = customize → newValue: !Null
operationType = update → newValue: Null
operationType = retrieval → newValue: Null
operationType = recreation → newValue: Null
syntax: alter(object, specifier, operation)
solution:
constraints (object, specifier, operation) → formal expression
dataType(object) = map, specifier = (((dom = v), rng)), operation = ((update, Null))
→ "override(~object, {v → recreation(~object(v))})"
specifier = field$_i$, operation = ((customize, nv))
→ " field$_i$ → nv"
specifier = field$_i$, operation = ((update, Null))
→ " field$_i$ → recreation(~object)"
dataType(object) = composite, specifier = (field$_i$), operation = ((customize, nv))
→ "modify(~object, field$_i$ → nv)"
dataType(object) = composite, specifier = (field$_i$), operation = ((update, Null))
→ "modify(~object, field$_i$ → recreation(~object.field$_i$))"
dataType(object) = composite, specifier = (field$_i$, field$_j$, …),
operation = ((operationType$_i$, newValue$_i$), (operationType$_j$, newValue$_j$), ⋯)
→ "modify(~object, alter(object, field$_i$, (operationType$_i$, newValue$_i$)),
alter(object, field$_j$, (operationType$_j$, newValue$_j$)), …)"
…

Fig. 2. Parts of the pattern "alter"

intermediate one. In the pattern *alter*, for example, the result is determined by the features of the three elements in *constituents* item. And except for formal notations, other kinds of expressions are also included in some results, such as "*recreation*()", as well as "*alter*()" which involves the pattern *alter* itself. These are actually reflecting the mechanism of reusing patterns which will be discussed in section 3.4.

Nevertheless, there exists a special pattern named *retrieve* that owns a different *solution* item because it is not able to build any expression-like intermediate result with the elements initially provided. It has only one element in the *constituent* item and needs to repeatedly obtain more information according to current situation and given rules until it is enough for the target expression generation. Because of this dynamic property, the content of the pattern is illustrated through its application process which will be presented in section 3.4.

3.2 Pattern Classification

Due to the inherent complexity of software, the number of patterns will be so large that the selection process becomes a hard task for users and the management would be difficult. To this end, we divide them into distinct categories.

With the experiences from many typical formal specifications, we found that almost all of them describe functions resulting from a combination of three kinds of basic functions: comparison between objects, acquisition of information and updating of existing data. Based on this consideration, patterns are classified into three categories: relation, retrieval and recreation.

Patterns of relation category provide a framework for describing relationships between objects. They are further classified into peer and non-peer ones. The former is used for depicting relations between objects of the same scope or kind while the latter is intended to present hierarchical relations. For example, the concept "bigger" is a possible relation between two integers or two sets of integers, but the description of a "bigger" relation between an integer and a set of integer makes no sense. Therefore, *bigger* is a peer relation pattern and a possible relation between an integer and a set of integer, such as $belongTo$, is identified as non-peer relation pattern.

There is only one pattern "*retrieve*" in the retrieval category. It is designed to help construct formal expressions denoting system variables. Although some of the system variables are already defined in formal specifications, but vast majority of them need to be represented by combinations of defined variables, such as the balance of one's account in a banking system. And the pattern *retrieve* is used to figure out such combinations to describe information acquisition.

Patterns of recreation deal with depicting system updating by presenting updates of system variables in formal notations. There are two kinds of updating described by different patterns, one is *common update* that updates data items of the intended variable, such as adding new data items, altering or deleting existing data items. The other is *specific update* that re-organizes or re-arranges data items of the intended variable by a given rule, such as sorting a set of data items into a sequence.

3.3 Structure of the Pattern System

In order to treat patterns as knowledge for our potential tool, they are organized into a pattern system preserving the classification strategy. Fig 3 shows the structure of the pattern system where texts in rectangles stand for category names while those in ellipse are referring to patterns. The semantics of this structure is the classification of the functions in real world explained in the previous section.

It should be noted that the given initial structure may be modified and expanded both in width and depth along with the design of new patterns and new mechanisms of the pattern system.

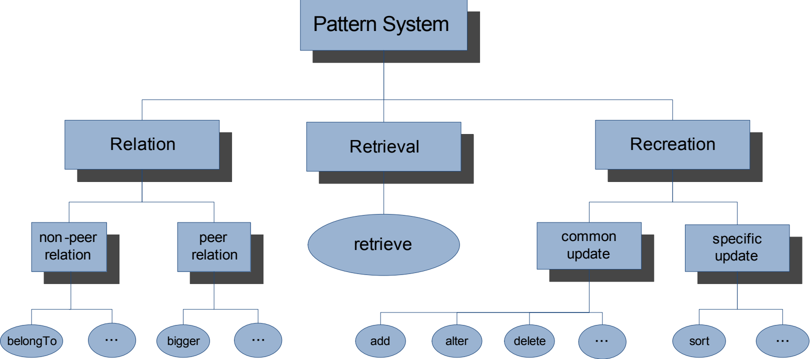


Fig. 3. The structure of the pattern system

3.4 Mechanism for the Application of the Pattern System

The pattern system can be used manually, but as we explained previously, to avoid the difficulties in studying, selecting and applying patterns by humans, we advocate an automated mechanism for the application of the pattern system. In this mechanism, patterns are selected based on the structure of the pattern system and each pattern is treated as a piece of knowledge that is used directly by computer to provide guidance for the user to efficiently apply it. Since the pattern *retrieve* is designed differently from patterns of relation and recreation, we discuss methods of applying them separately.

Pattern selection. When users need to express informal ideas in formal notations, they would have to first clarify the general intention. This task is fulfilled by selecting an appropriate pattern which can be guided along the hierarchical structure of the pattern system on the semantic level. Distinct identification of each category and pattern simplifies decision making and *explanation* item of each pattern help confirm the correctness of the selection.

Application of patterns of relation and recreation. Applying patterns of relation and recreation largely depends on their structures and is therefore straightforward. Once a pattern is selected, the user will be asked to provide elements according to the *constituents* item, and then a suggested formal fragment will be generated based on the given elements and the *solution* item. Besides, these patterns can also be invoked by expressions written consistent with the *syntax* item. The expression will be analyzed to check whether the provided elements are all choice elements or represented by defined variables, and whether each required element is available. If this is true, a suggested formal fragment can be given, otherwise, complement or revision is needed.

But in most cases, obtaining a formal description can not be done in one step. Regarding the maintainability and the generation of effective guidance as our goals, the reuse mechanism is introduced which was mentioned in the previous

section. It occurs during the application of patterns that involve names of reused patterns or categories in certain mapped results of their *solution* items. Each reused pattern name indicates the application of the pattern and each category name denotes the application of the pattern selected within the category under guidance, which refines these intermediate results. The final formal fragment can then be derived by repeating the above process until no reused pattern or category is involved. For example, in the *solution* item of the pattern *alter*, the word "*alter*" refers to the reuse of the pattern *alter* itself, and the word "*recreation*" indicates that the user will be guided to select a specific pattern of recreation category to apply it. Whether such reuse will occur depends on how the element $operationType$ is specified which is composed of four children choice elements. If *customize* has been chosen, which means that the user would like to provide a concrete value as the updated one, $newValue$ will be designated as the given value without reusing any pattern. Selection on one of the other children elements, however, tells that the retrieval of the new value needs help resulting in application of other patterns. Choice element *retrieve* is used to activate the pattern *retrieve* while *recreation* and *update* indicate applying recreation patterns on another variable and the original value respectively.

Often, reused patterns or categories appear in *solution* items are associated with element information. For patterns the information is the elements required in its *constituents* item written consistent with *syntax*. Reused categories can also temporarily hold the information for the later selected pattern to use, such as the included "$recreation(\tilde{~}object(v))$" in the *solution* item of the pattern *alter*. It demonstrates that when a specific pattern p belonging to recreation category is chosen under guidance, $\tilde{~}object(v)$ will be transferred to p as the value of the first element according to its *syntax.*

Application of the pattern *retrieve*. Supporting to describe system variables in formal expressions, the pattern *retrieve* owns only one element $initialType$ in the *constituent* item and holds a different application method with others because of its special *solution* item. Rather than mappings from features to formal fragments, the *solution* item of the pattern *retrieve* gives a process for deriving the target formal expression with the given $initialType$. The main strategy is to construct a tree structure to collect necessary information from users and generate a formal expression based on the tree.

As the essential of the approach, the tree uses left and right subtrees to present constraints on the intended variable represented by the root node. In such structure, each node is identified as a defined type while each branch $branch_i$ is represented as a transition (s, l, s') where s' is a child node of s, l is the label of the branch. Branches are divided into two kinds. If s' is a subtype of s, i.e., the definition of s relies on the definition of s'(denoted as $Dep(s, s')$), $branch_i$ is a downward branch where s' is a left child of s. If $Dep(s', s)$ establishes, $branch_i$ is a upward branch where s' is a right child of s. Downward branches use constraints on s' as their labels while upward branches take constraints on s as their labels. For example, suppose an informal idea is to retrieve a system variable obj that satisfies two conditions: the data type of obj is $real * int$ and $obj(2) = 5$. It

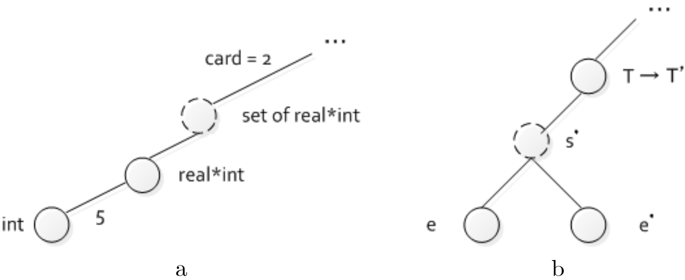


Fig. 4. Pseudo node for special situations

can be represented as a downward branch $(real*int, l_1, int)$ where l_1 is set to be "5" as the constraint on node *int*. In the case that the intended variable *obj* is an element of a set declared as *set of int* that satisfies $obj > 5$, we should use a upward branch $(int, l_2, set\ of\ int)$ where l_2 is set to be "> 5" as the constraint on node *int*.

Nodes of the left subtree can only own downward branches while that of the right subtree are able to own both kinds of branches. And the rightmost leaf node of the right subtree corresponds to a defined variable serving as the basis of the target formal expression. Let's take the previous upward branch $(int, l_2, set\ of\ int)$ as an example, assume that node *set of int* corresponds to a defined variable v, *obj* can then be presented as $(obj\ inset\ v) \wedge (obj > 5)$.

Besides, pseudo node is introduced to handle special situations. One situation is that certain node corresponds to more than one system variable. In the previous example, there might be a set of obj_i satisfying $obj_i(2) = 5$. Furthermore, the user may want to present a condition that the nodes of upper levels should satisfy by giving constraints on this set. For example, the intended variable becomes a sequence of product containing 2 obj_i where $obj_i(2) = 5$. To enable such kind of description in the tree structure, we create a pseudo node *set of real∗int* as shown in Fig 4(a). But if the root node happens to be in such situation, it will be turned into a pseudo node without creating a new one. The other situation is that some constraints have to be defined by composite values. For example, a node s identified as $T \rightarrow T'$ may be required to satisfy that one of the elements e in $dom(s)$ maps to the element e' in $rng(s)$. To express such meaning, a pseudo node $s\prime$ will be created as shown in Fig 4(b).

Based on the above concepts, the construction method of the tree can be given which only requires users' decisions on the semantic level and is therefore easy. During this process, developing branches is a critical operation that needs to be presented first. For a node s, its downward branches $branchD(s)$ are developed by setting selected subtypes of s as left child nodes and attaching given constraints on these subtypes as labels. By contrast, its upward branch $branchU(s)$ is only one new branch (s, l, t) where t is the selected type that takes s as its subtype, l is the given constraints on s. If s encounters special situations, pseudo nodes will be created following the instructions mentioned above.

The construction process starts from the root node s_0 which stands for the value of element *initialType* meaning the data type of the intended variable. By generating downward branches for each node s that has been currently extended to, the left subtree will be built. In case that s_0 cannot develop downward branch, the left subtree will be empty. However, the right subtree can always be built by the following algorithm.

1. Generate $branchU(s_0)$ and set the current leaf node as the current node cn.
2. If cn is a right child node of the right subtree and there exists a defined variable v of type cn confirmed and accepted by the user, then quit with v.
3. If cn needs to be identified by constraints from its subtypes, then extend the left subtree of cn using the proposed method for left subtree generation.
4. If cn is not a left child of certain node of the right subtree, then generate $branchU(cn)$.
5. Set each current leaf nodes as the current node and repeat 2 – 5 respectively.

With a complete tree, the target formal expression exp is generated by treating the left and right subtree separately. The expression $lExp$ standing for the left subtree is obtained through the following algorithm, which can be skipped for the trees with empty left subtrees:

```
create a stack currentNodes;
currentNodes.push(s_0);
while(currentNodes is not empty){
  currentNode = currentNodes.pop();
  lExp = merge(Null, lExp, branchD(currentNode));
    for each child child_i of currentNode
       {currentNodes.push(child_i); }}
```

And the algorithm to form the expression $rExp$ standing for the right subtree is as follows where $rLeaf$ denotes the leaf node of the right subtree:

```
currentNode = rLeaf.parentNode;
rExp = v;
while(currentNode! = s_0){
   tempExp = Null;
    if(currentNode has left subtree){
        create a stack temps;
        temps.push(currentNode);
        temp = currentNode;
        while(temps is not empty){
          currentNode = temps.pop();
          tempExp = merge(Null, tempExp, branchD(currentNode));
           for each child tempChild_i of currentNode
              {temps.push(tempChild_i); }}}
    rExp = merge(rExp, tempExp, {branchU(temp)});
    currentNode = temp.parentNode; }}
```

Left subtrees are dealt with in a top-down manner while the right subtrees are transformed by a bottom-up method with the critical variable v as the start point. Finally after combining two expressions through the root node, the final formal expression is achieved as:

$exp = merge(rExp, lExp, \{branchU(s_0)\})$

Along the whole transformation process, function $merge$ plays an important role which is defined as:

$merge : rExp * lExp* \ set \ of \ branch \rightarrow exp$

where $rExp$ is a string denoting an expression transformed from a right subtree, $lExp$ is a string denoting an expression transformed from a left subtree, and exp is a string denoting the expression generated by combining $rExp$, $lExp$ and a set of branches. It is designed for constructing expressions under various situations, but the detailed rules are not further discussed for the sake of space.

4 Case Study

We apply the proposed pattern system to a banking system for accounts management. Each account is owned by one customer with a unique pair of account number and password, containing information of balance and transactions. Balance is a set of maplets from a currency type to its current amount and transactions is a sequence of transaction, each recording the time, type, currency type

```
type
AccountNo = seq of nat0;
Password = string;
CustomerInf = composed of
                  accountNo: AccountNo
                  password: Password
                 end;
CurrencyType = {<USD>, <JPY>, <CNY>};
Amount = real;
Balance = map CurrencyType to Amount
Year = nat0;
Month = nat0;
Day = nat0;
Date = Year*Month*Day;
OperationType = {<deposit>, <withdraw>}
Transaction = composed of
                 date: Date
                 operationType: OperationType
                 currencyType: CurrencyType
                 amount: Amount
                end;
Transactions = seq of Transaction;
AccountInf = composed of
                balance: Balance
                transactions: Transactions
               end;
AccountFile = map CustomerInf to AccountInf;
var
ext #account_store: AccountFile
```

Fig. 5. Definitions of types and variables of the example specification

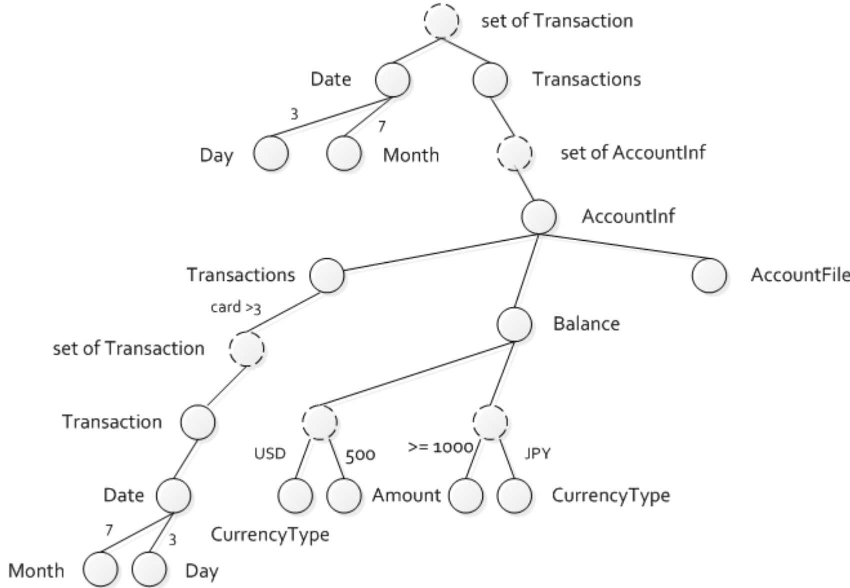


Fig. 6. The tree structure for the example display function

```
let accountInfSet = {item_AccountInf | item_AccountInf : rng(account_store)
                      & let transactionSeq = [item_Transaction |
                             item_Transaction : Transaction
                             & exists[i : int] | item_AccountInf.transactions(i) = item_Transaction
                                   and item_Transaction.date(2) = 7 and item_Transaction.date(3) = 3]
                        in len(transactionSeq) > 3
                        and item_AccountInf.balance(USD) = 500
                        and item_AccountInf.balance(JPY) >= 1000}
in  let TransactionsSet = {item_Transactions | item_Transactions :Transactions
                            & exists[item_AccountInf : accountInfSet] |
                              item_AccountInf.transactions = item_Transactions}
    in  output = {item_Transaction | item_Transaction : Transaction
                   & exists[item_Transactions : TransactionsSet, j: int] |
                     item_Transactions(j) = item_Transaction}
                   and item_Transaction.date(2) = 7 and item_Transaction.date(3) = 3
```

Fig. 7. The formal expression of the example display function

and amount of certain operation. All of the information mentioned above is kept in an external store defined as *account_store*. Accordingly, we assume that the *type* and *var* parts of the specification are defined as in Fig 5.

Main functions of this example banking system includes *customer authentication*, *information display, deposit* and *withdraw*. Considering that *deposit* has the same nature with *withdraw*, we use *withdraw* to illustrate both of them.

Customer authentication checks whether the given customer information, denoted as *customerInf*, belongs to the authorized data store. This is apparently

name: add
constituents: object, addend
(object, addend)
dataType(object) = set of T: composed of
$field_1 : T_1$
…
$field_n : T_n$
→ addend : $v_1 \wedge \ldots \wedge v_n$
…
syntax: add(object, addend)
solution:
constraints(object, addend) → formal expression
dataType(object) = set of T: composed of
$field_1 : T_1$
…
$field_n : T_n$
∧ addend = (v1, v2,⋯v_n)
→ "conc(object, mk_T(v1, v2,⋯v_n))"
…

name: delete
constituents: object, minuend
(object, minuend)
dataType(object) = real
→ minuend : v
dataType(object) = set of T: composed of
$field_1 : T_1$
…
$field_n : T_n$
→ minuend : $field_1 \wedge \ldots \wedge field_n$
…
syntax: delete(object, minuend)
solution:
constraints(object, minuend) → formal expression
dataType(object) = real
∧ minuend = v → "object - v"
…

Fig. 8. Parts of the pattern "add" and "delete"

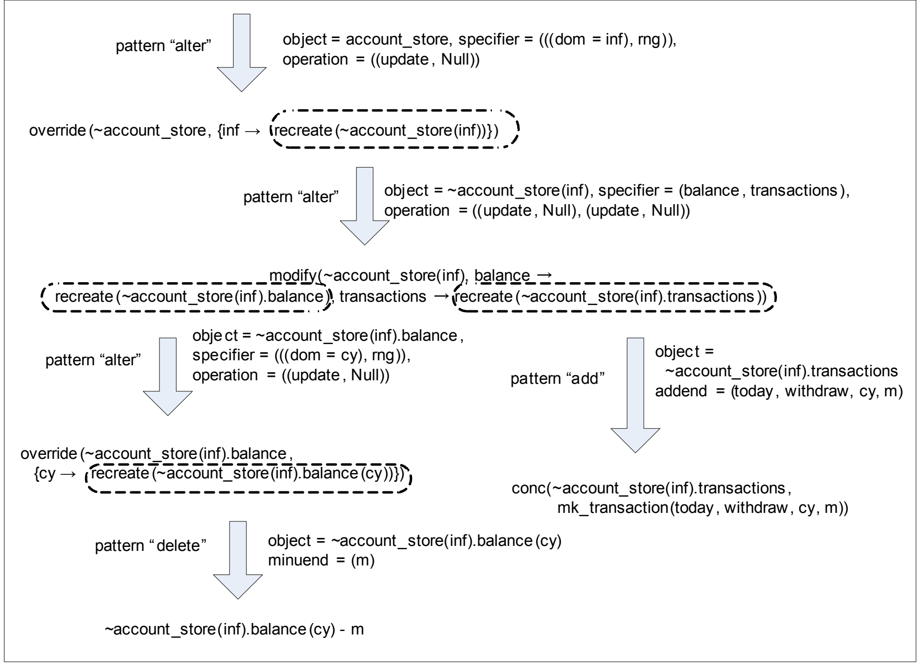


Fig. 9. Generation process of the formal expression representing the updated data store in the example function "withdraw" using pattern "alter"

an issue about relation, which leads to the selection of relation category in the top level of the pattern system structure. Through deeper explorations in the structure, pattern *belongTo* becomes the most suitable one. According to its

definition, two elements *element* and *container* must be specified. In our case, they should be designated as $customerInf$ and $account_store$ respectively. These two variables will be automatically analyzed in the context of the *solution* item, which results in a formal expression "$customerInf\ inset\ dom(account_store)$" to describe the relation.

Information display obtains data required by customers, which falls into the scope of the pattern *retrieve*. To better demonstrate the application of the pattern, we complicate the function on purpose. Consider describing displaying the July 3rd's transactions of accounts that have more than 3 transaction records on July 3rd, 500 US dollars and more than 1000 Japanese Yen. By applying the pattern *retrieve*, a tree structure(shown in Fig 6) can be constructed. Using the proposed transformation method, we will get the result formal expression as shown in Fig 7 where $output : set\ of\ Transaction$ denotes the output variable.

The core of the *withdraw* function is the updating of the external data store $account_store$ which needs the pattern *alter* to help describe. Suppose a customer with information inf has withdrawn certain amount m of certain kind of currency cy, through the process shown in Fig 9, the expression representing the updated data store is generated as follows(Due to the reuse of pattern *add* and *delete*, parts of them are presented in Fig 8 as a reference):

$$
\begin{aligned}
&.override(\tilde{}account_store,\\
&\ \{inf \rightarrow modify(\tilde{}account_store(inf),\\
&\quad balance \rightarrow override(\tilde{}account_store(inf).balance,\\
&\qquad\qquad \{cy \rightarrow \tilde{}account_store(inf).balance(cy) - m\}),\\
&\qquad\qquad\ transactions \rightarrow conc(\tilde{}account_store(inf).transactions,\\
&\qquad\qquad\qquad\qquad mk_transaction(today, withdraw, cy, m))\})
\end{aligned}
$$

5 Conclusion

This paper describes a pattern system to support refining informal ideas into formal expressions. The pattern system organizes patterns into a hierarchical structure according to different properties of individual patterns, which greatly facilitates pattern selection. Moreover, users will be explicitly guided by application methods of the pattern system without the need of understanding it.

In the future, we are interested in exploring more patterns for type and state variable declarations, predicate expression construction, and specification architecture construction. Another important research will be building a software tool to support the pattern system presented in this paper.

References

1. Silverstein, M., Alexander, C., Ishikawa, S.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, Oxford (1977)
2. Miller, A., Manolescu, D., Kozaczynski, W.: The growing divide in the patterns world. IEEE Software 24(4), 61–67 (2007)
3. Johnson, R., Gamma, E., Helm, R.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, Reading (1994)

4. Grunske, L.: Specification Patterns For Probabilistic Quality Properties. In: Proceedings of the 30th International Conference on Software Engineering, pp. 31–40 (2008)
5. He, X., Ding, J., Mo, L.: An approach for specification construction using property-preserving refinement patterns. In: 23th Annual ACM Symposium on Applied Computing, pp. 797–803. ACM, New York (2008)
6. Kurita, T., Nakatsugawa, Y., Ohta, Y.: Applying Formal Specification Method in the Development of an Embedded Mobile FeliCa IC Chip. In: Proceedings of the 2005 Software Symposium, Japan, pp. 73–80 (June 2005) (in Japanese)
7. Liu, S.: Formal Engineering for Industrial Software Development. Springer, Heidelberg (2004)
8. Liu, S., Takahashi, K., Hayashi, T., Nakayama, T.: Teaching Formal Methods in the Context of Software Engineering. In SIGCSE Bulletin 41(2), 17–23 (2009)
9. Corbett, J.C., Dwyer, M.B., Avrunin, G.S.: Pattern in property specifications for finite-state verification. In: 21th International Conference on Software Engineering, pp. 411–420. ACM, New York (1999)
10. Parnas, D.L.: Really Rethinking Formal Methods. Computer 43(1), 28–34 (2010)
11. Sahara, S.: An Experience of Applying Formal Method on a Large Business Application. In: Proceedings of 2004 Symposium of Science and Technology on System Verification, Osaka, Japan, February 4-6, pp. 93–100. National Institute of Advanced Industrial Science and Technology (AIST) (2004) (in Japanese)
12. Stepney, S., Polack, F., Toyn, I.: An outline pattern language for Z: five illustrations and two tables. In: Bert, D., Bowen, J.P., King, S. (eds.) ZB 2003. LNCS, vol. 2651, pp. 2–19. Springer, Heidelberg (2003)
13. The VDM-SL Tool Group: Users Manual for the IFAD VDM-SL tools. Technical Report IFAD-VDM-4, The Institute of Applied Computer Science (IFAD) (December 1994)
14. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys 41(4) (2009)

Specification Translation of State Machines from Equational Theories into Rewrite Theories

Min Zhang[1], Kazuhiro Ogata[1], and Masaki Nakamura[2]

[1] School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
{zhangmin,ogata}@jaist.ac.jp
[2] School of Electrical and Computer Engineering
Kanazawa University
masaki-n@is.t.kanazawa-u.ac.jp

Abstract. Specifications of state machines in CafeOBJ are called equational theory specifications (EQT Specs) which are based on equational logic, and in Maude are called rewrite theory specifications (RWT Specs) which are based on rewriting logic. The translation from EQT Specs to RWT Specs achieves the collaboration between CafeOBJ's theorem proving facilities and Maude's model checking facilities. However, translated specifications by existing strategies are of inefficiency and rarely used for model checking in practice. This paper defines a specific class of EQT Specs called EADS Specs, and proposes a strategy for the translation from EADS Specs to RWT Specs. It is proved that translated specifications by the strategy are more efficient than those by existing strategies.

Keywords: Algebraic specification, automatic translation, rewrite theory, equational theory, CafeOBJ, Maude.

1 Introduction

Specification translation is a traditional way of achieving the collaboration between different verification tools, with duplicate effort reduced at the specification level. Translations between different formalisms have been widely studied. For instance, the translation from Z into B [1] integrates the tool PROZ for Z specifications into PROB; safe Petri Nets are translated into statecharts to enable the automated exchange of models between Petri net and statechart tools [2]; and Raise Specification Language (RSL) is translated into CSPM so that LTL formulae in RAISE can be model checked by the model checker FDR [3].

CafeOBJ [4] and Maude [5] are two state-of-the-art verification systems based on algebraic approaches. CafeOBJ is equipped with theorem proving facilities [6], while Maude with model checking facilities. Whenever a property fails to be proved in CafeOBJ, a counterexample is desired. In this situation, Maude is a better alternative than other model checking tools for the following reasons: (1) it is a sister language of CafeOBJ and has similar syntax, which reduces duplicate effort at specification level, and (2) the efficiency of Maude model checking facilities is comparable to those of other prevalent tools like SPIN [7].

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 678–693, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Specifications of state machines in CafeOBJ are equational theory specifications (EQT Specs), and in Maude are rewrite theory specifications (RWT Specs). There are multiple styles of equational theory or rewrite theory specifications of state machines. EQT Specs in this paper only refer to a class of specifications that are developed in OTS/CafeOBJ method [8], and RWT Specs to a sub-class of rewrite theory specifications where states are represented by sets of observable components and action components (see Section 2.3 for details). Automatic translation from EQT Specs to RWT Specs is much more preferable because manually developing an RWT Spec for the state machine that is specified by an EQT Spec is not only effort-consuming, but at risk of causing inconsistencies between the EQT Spec and the RWT Spec. Recently, studies on the specification translation of state machines between the two formalisms have been conducted. Three strategies have been proposed so far to automate the translation and translators have been developed [9,10,11]. However, specifications generated by these strategies are rarely used in practice for model checking due to the low efficiency of the translated specifications.

This paper proposes a translation strategy for a specific sub-class of EQT Specs, aiming at generating more efficient model checkable RWT Specs. We argue that not all EQT Specs can be translated into RWT Specs, and hence introduce a specific class of EQT Specs called EADS Specs from a practical point of view. EADS Specs are mainly used to specify a class of asynchronous systems called Extended Asynchronous Distributed Systems (EADS). We compare the efficiencies of translated specifications that are obtained in different strategies with two concrete examples. The experimental result indicates that the efficiency of translated specifications is significantly improved. The contributions of this work are manifold: (1) a specific class (EADS Specs) of EQT Specs that are used for practical verifications are discovered; (2) a translation strategy is proposed to automate the translation from EADS Specs into RWT Specs; and (3) the efficiency of translated specifications is significantly improved so that they can be used by Maude for practical model checking.

The rest of this paper is organized as follows: Section 2 introduces state machines, EQT Specs and RWT Specs. Section 3 introduces EADS Specs and explains the reason why EADS Specs are selected. Section 4 describes a strategy for the translation from EADS Specs into RWT Specs. The efficiency of the specifications generated by our strategy is evaluated through comparing with those generated by three existing strategies in Section 5. Section 6 concludes this paper and mentions ongoing work.

2 Preliminaries

2.1 State Machines

A state machine consists of (1) a set $\mathcal{U}$ of states, (2) the set $\mathcal{I}(\mathcal{I} \subseteq \mathcal{U})$ of initial states, and a set $\mathcal{T}$ of transitions. Each $u \in \mathcal{U}$ is a (possibly infinite) record $\{l_1 = d_1, l_2 = d_2, \ldots\}$ of type $\{l_1 : D_1, l_2 : D_2, \ldots\}$, where D with a subscript such as D_i is a type for data. For convenience, we let $l_i(u)$ denote l_i's

corresponding value d_i in state u. Each transition $t \in \mathcal{T}$ is a binary relation over states.

Let us consider a mutual exclusion protocol called *Qlock* to show how to model dynamic systems with state machines. Multiple processes participate in Qlock and each process p executes the following program:

Loop
 rs: *enqueue(queue,p);*
 ws: **repeat until** *top(queue) = p;*
  critical section;
 cs: *dequeue(queue);*

The *queue* records the processes that are requesting to enter the critical section according to the request order. Initially, all processes are at label rs, and the shared queue is empty. A process p puts its process identifier at the bottom of the queue and then waits to enter the critical section. It is allowed to enter the critical section whenever its identifier is at the top of the queue and it is at the label ws. The process executes the *dequeue* operation on the shared queue when it leaves the critical section.

Let *Queue*, *Label* and *Pid* be types respectively for queues of process identifiers, labels (rs, ws and cs) and process identifiers $(p_1, p_2, \ldots)$. A state machine $\mathcal{M}_{\text{Qlock}}$ modeling Qlock is as follows:

- $\mathcal{U}_{\text{Qlock}} \triangleq \{u | u : \{queue : Queue, pc_1 : Label, pc_2 : Label, \ldots\}\}$;
- $\mathcal{I}_{\text{Qlock}} \triangleq \{u_0 \in \mathcal{U}_{\text{Qlock}} | queue(u_0) = empty, pc_i(u_0) = rs \text{ } for \text{ } each \text{ } process \text{ } p_i\}$;
- $\mathcal{T}_{\text{Qlock}} \triangleq \{want_1, want_2, \ldots\} \cup \{try_1, try_2, \ldots\} \cup \{exit_1, exit_2, \ldots\}$.
  - $(u, u') \in want_i$ iff $pc_i(u) = rs, pc_i(u') = ws, queue(u') = (p_i | queue(u))$ and $pc_j(u') = pc_j(u)$ for each process p_j s.t. $p_j \neq p_i$;
  - $(u, u') \in try_i$ iff $pc_i(u) = ws, top(queue(u)) = k, pc_i(u') = cs, queue(u) = queue(u')$ and $pc_j(u') = pc_j(u)$ for each $p_j \neq p_i$;
  - $(u, u') \in exit_i$ iff $pc_i(u) = cs, queue(u') = dequeue(queue(u)), pc_i(u') = rs$ and $pc_j(u') = pc_j(u)$ for each $p_j \neq p_i$.

Fig. 1 shows a part of state transitions in $\mathcal{M}_{\text{Qlock}}$. An arrow labelled by a transition t from a state u to u' denotes $(u, u') \in t$. Elements in queues are concatenated by $|$. The rightmost element is taken as the top one in the queue.

2.2 EQT Specs

EQT Specs are based on equational logic in the sense that transitions are specified with a set of equations. Let Υ be a sort for states. An EQT Spec consists of (1) a finite set $\mathcal{O}$ of observers, (2) a constant *init* of Υ, representing an arbitrary initial state, (3) a finite set $\mathcal{A}$ of actions, and (4) a family $\mathcal{E}$ of sets of equations. Each observer o is a function symbol whose rank is $\Upsilon \ D_{o1} \ldots D_{om} \rightarrow D_o$. An observer corresponds to a (possibly infinite) set of data fields in a state in state machines. Each action a is a function symbol whose rank is $\Upsilon \ D_{a1} \ldots D_{an} \rightarrow \Upsilon$. An action a represents a (possibly infinite) set of transitions in state machines.

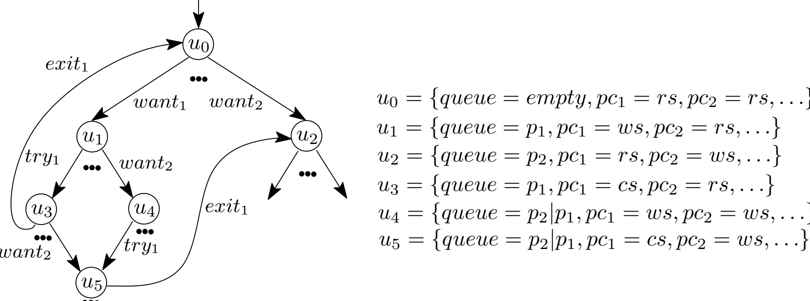


Fig. 1. State transitions in the state machine of *Qlock*

Each action a is given a function symbol c-a with the same arity of a and Bool as its coarity, denoting the condition under which a transition represented by action a takes place. $\mathcal{E}$ consists of a set $\mathcal{E}_{init}$ of equations which *init* must satisfy, and a set $\mathcal{E}_a$ of equations for each action a which can be interpreted as the definition of a set of transitions denoted by a.

We take an EQT Spec $\mathcal{S}_{\text{Qlock}}$ for $\mathcal{M}_{\text{Qlock}}$ as an example. Let `Pid`, `Queue`, and `Label` be sorts for process identifiers, queues and labels[1]. Function symbols `enqueue`, `dequeue` and `top` correspond to basic functions *enqueue*, *dequeue* and *top* on type Queue. Constants `rs`, `ws` and `cs` are of sort `Label`, corresponding to labels rs, ws and cs, respectively.

$\mathcal{S}_{\text{Qlock}}$

$\mathcal{O}_{\text{Qlock}} \triangleq \{\texttt{pc} : \Upsilon\ \texttt{Pid} \to \texttt{Label}, \texttt{queue} : \Upsilon \to \texttt{Queue}\}$;
$\mathcal{A}_{\text{Qlock}} \triangleq \{\texttt{want} : \Upsilon\ \texttt{Pid} \to \Upsilon, \texttt{try} : \Upsilon\ \texttt{Pid} \to \Upsilon, \texttt{exit} : \Upsilon\ \texttt{Pid} \to \Upsilon\}$;
$\mathcal{E}_{\text{Qlock}} \triangleq \{\mathcal{E}_{\texttt{init}}, \mathcal{E}_{\texttt{want}}, \mathcal{E}_{\texttt{try}}, \mathcal{E}_{\texttt{exit}}\}$, where:

- $\mathcal{E}_{\texttt{init}} \triangleq \{\texttt{pc}(\texttt{init}, x) = \texttt{rs}, \texttt{queue}(\texttt{init}) = \texttt{empty}\}$
- $\mathcal{E}_{\texttt{want}} \triangleq \{\texttt{c-want}(\upsilon, y) = \texttt{pc}(\upsilon, y) \doteq \texttt{rs}$
  $\texttt{pc}(\texttt{want}(\upsilon, y), x) = (\textbf{if } x \doteq y \textbf{ then } \texttt{ws} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi}) \textbf{ if } \texttt{c-want}(\upsilon, y)$
  $\texttt{queue}(\texttt{want}(\upsilon, y)) = (y|\texttt{queue}(\upsilon)) \textbf{ if } \texttt{c-want}(\upsilon, y)$
  $\texttt{want}(\upsilon, y) = \upsilon \textbf{ if } \neg\texttt{c-want}(\upsilon, y)\}$
- $\mathcal{E}_{\texttt{try}} \triangleq \{\texttt{c-try}(\upsilon, y) = \texttt{pc}(\upsilon, y) \doteq \texttt{ws} \land \texttt{top}(\texttt{queue}(\upsilon)) \doteq y$
  $\texttt{pc}(\texttt{try}(\upsilon, y), x) = (\textbf{if } x \doteq y \textbf{ then } \texttt{cs} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi}) \textbf{ if } \texttt{c-try}(\upsilon, y)$
  $\texttt{queue}(\texttt{try}(\upsilon, y)) = \texttt{queue}(\upsilon) \textbf{ if } \texttt{c-try}(\upsilon, y)$,
  $\texttt{try}(\upsilon, y) = \upsilon \textbf{ if } \neg\texttt{c-try}(\upsilon, y)\}$
- $\mathcal{E}_{\texttt{exit}} \triangleq \{\texttt{c-exit}(\upsilon, y) = \texttt{pc}(\upsilon, y) \doteq \texttt{cs}$
  $\texttt{pc}(\texttt{exit}(\upsilon, y), x) = (\textbf{if } x \doteq y \textbf{ then } \texttt{rs} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi}) \textbf{ if } \texttt{c-exit}(\upsilon, y)$
  $\texttt{queue}(\texttt{exit}(\upsilon, y)) = \texttt{dequeue}(\texttt{queue}(\upsilon)) \textbf{ if } \texttt{c-exit}(\upsilon, y)$
  $\texttt{exit}(\upsilon, y) = \upsilon \textbf{ if } \neg\texttt{c-exit}(\upsilon, y)\}$

---

where υ is a variable of Υ, and x, y are of `Pid`. Symbol $\doteq$ denotes equivalence relations over data types. Every variable in an equation (or a rewriting rule) is universally quantified and its scope is in the equation (or the rewriting rule). The observer `queue` specifies the field *queue* : *Queue* in the record type in

[1] By convention, symbols like sorts, constants and function symbols at specification level are differentiated from those at mathematical level by using typewriter font.

$\mathcal{M}_{\text{Qlock}}$, and `pc` specifies an infinite set of fields $\{pc_1 : Label, pc_2 : Label, \ldots\}$. Constant `init` together with $\mathcal{E}_{\texttt{init}}$ specifies $\mathcal{I}_{\text{Qlock}}$. Action `want` corresponds to an infinite set $\{want_1, want_2, \ldots\}$ of transitions. The set $\mathcal{E}_{\texttt{want}}$ of equations can be interpreted as the definition of the set of transitions. Actions `try` and `exit` together with $\mathcal{E}_{\texttt{try}}$ and $\mathcal{E}_{\texttt{exit}}$ specify the sets of transitions $\{try_1, try_2, \ldots, \}$ and $\{exit_1, exit_2, \ldots, \}$. Term $\texttt{want}(v, y)$ represents a successor of the state denoted by v if $\texttt{c-want}(v, y)$ holds. Otherwise, $\texttt{want}(v, y)$ is considered equivalent to v.

States in $\mathcal{M}_{\text{Qlock}}$ are represented by terms of Υ. For example, states u_0, u_1, and u_4 as shown in Fig. 1 are represented by terms `init`, $\texttt{want}(\texttt{init}, p_1)$, and $\texttt{want}(\texttt{want}(\texttt{init}, p_1), p_2)$, respectively. Taking u_1 for instance, we have $pc_1(u_1) = ws$. Term $\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_1)$ equals `ws` for the following reasons. According to the second equation in $\mathcal{E}_{\text{want}}$ with v being `init`, x and y being p_1, we have

$$\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_1) = (\textbf{if } p_1 \doteq p_1 \textbf{ then } \texttt{ws} \textbf{ else } \texttt{pc}(\texttt{init}, p_1) \textbf{ fi})$$
$$\textbf{if } \texttt{c-want}(\texttt{init}, p_1)$$

According to the first equation in $\mathcal{E}_{\texttt{init}}$, $\texttt{pc}(\texttt{init}, p_1)$ is equivalent to `rs`, which indicates $\texttt{c-want}(\texttt{init}, p_1)$ holds. Because $p_1 \doteq p_1$ holds, the right-hand side (RHS) of the equation above equals `ws`, namely that $\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_1)$ equals `ws`. Similarly, we have $\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_i)$ equals `rs` for $p_i (i > 1)$ and $\texttt{queue}(\texttt{want}(\texttt{init}, p_1))$ equals p_1.

2.3 RWT Specs

RWT Specs are based on rewriting logic in the sense that transitions are specified by rewriting rules. A state is represented as a set of components denoted by sort `State`. Components in a state can be divided into two kinds, namely *action components* and *observable components* whose sorts are `AComp` and `OComp` as subsorts of `State`. Each action component corresponds to a set of transitions in state machines, and each observable component to a data field in a state.

An RWT Spec consists of (1) a finite set $\mathcal{OC}$ of observable component constructors, (2) a finite set $\mathcal{AC}$ of action component constructors, (3) a set $\mathcal{F}$ of function symbols for the representation of initial states, with a set $\mathcal{E}_{\mathcal{F}}$ of equations for the function symbols in $\mathcal{F}$, and (4) a finite set $\mathcal{R}$ of rewriting rules. Each observable component constructor $o[_, \ldots, _]{:}_$ is a function symbol whose rank is $D_{o1} \ldots D_{om}\ D_o \rightarrow \texttt{OComp}$. We adopt mixfix operators in CafeOBJ and Maude. An underscore indicates the place where an argument is put. An observable component constructor corresponds to a (possibly infinite) set of fields of record type (i.e. the type of states) in a state machine. An observable component is expressed by a term whose top is $o[_, \ldots, _]{:}_$. Each action component constructor ac is a function symbol whose rank is $SetD_{t1} \ldots SetD_{tn} \rightarrow \texttt{AComp}$, where $SetD_{ti}$ is a sort for sets of elements of D_{ti}[2]. An action component is expressed by a term

[2] Basic operations on $SetD_{tk}$ follow the definition of basic set in [5, chap. 5]. Whitespace character is defined as concatenation operation which is associative and commutative. Therefore, variable y_k of D_{tk} can be any element of D_{tk} in a pattern $(y_k\ ys_k)$, where variable ys_k is of $SetD_{tk}$.

whose top is ac, corresponding to a set of transitions in state machines. One rewriting rule in $\mathcal{R}$ specifies a (possibly infinite) set of transitions. For instance, the following rewriting rule specifies all transitions in $\{want_1, want_2, \ldots\}$

$$\texttt{want}((y\ ys))\ (\texttt{queue:}\ q)\ (\texttt{pc[}y\texttt{]:}\ l) \Rightarrow$$
$$\texttt{want}((y\ ys))\ (\texttt{queue:}\ (y|q))\ (\texttt{pc[}y\texttt{]:}\ \texttt{ws})\ \textbf{if}\ l \doteq \texttt{rs},$$

where, q is a variable of sort `Queue` and l of `Label`. A state containing the fields $queue = q'$ and $pc_i = rs$ is partially denoted by $\texttt{want}(p_i\ ys)(\texttt{queue:}\ q)(\texttt{pc[}y\texttt{]}:l)$. The term is rewritten into $\texttt{want}(p_i\ ys)(\texttt{queue:}\ (p_i|q'))(\texttt{pc[}p_i\texttt{]}:\texttt{ws})$, which means that the two corresponding data fields in a state are changed into $queue = (p_i|q')$ and $pc_i = ws$. The rewriting rule says that if there is a process y whose label is `rs` and a queue is q in a state, there is a successor state where the label of process y becomes `ws` and the queue $(y|q)$.

An RWT Spec $\mathfrak{S}_{\mathrm{Qlock}}$ that specifies the state machine $\mathcal{M}_{\mathrm{Qlock}}$ is as follows:

$\mathfrak{S}_{\mathrm{Qlock}}$

$\mathcal{OC}_{\mathrm{Qlock}} \triangleq \{\texttt{pc[_]:_: Pid Label} \to \texttt{OComp}, \texttt{queue:_: Queue} \to \texttt{OComp}\}$;

$\mathcal{AC}_{\mathrm{Qlock}} \triangleq \{\texttt{want: SetPid} \to \texttt{TComp}, \texttt{try: SetPid} \to \texttt{TComp}, \texttt{exit: SetPid} \to \texttt{TComp}\}$;

$\mathcal{F}_{\mathrm{Qlock}} \triangleq \{\texttt{init: SetPid} \to \texttt{State}, \texttt{mk-pc: SetPid} \to \texttt{State}\}$;

$\mathcal{E}_{\mathcal{F}_{\mathrm{Qlock}}} \triangleq \{\texttt{init}(ys) = \texttt{want}(ys)\ \texttt{try}(ys)\ \texttt{exit}(ys)\ (\texttt{queue: empty})\ \texttt{mk-pc}(ys),$
$\texttt{mk-pc(empty-set)} = \texttt{empty-state},$
$\texttt{mk-pc}(y\ ys) = (\texttt{pc[}y\texttt{]: rs})\ \texttt{mk-pc}(ys)\}$

$\mathcal{R}_{\mathrm{Qlock}} \triangleq \{rw_{\texttt{want}}, rw_{\texttt{try}}, rw_{\texttt{exit}}\}$, where:

- $rw_{\texttt{want}} \triangleq \texttt{want}((y\ ys))(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow$
  $\texttt{want}((y\ ys))(\texttt{pc[}y\texttt{]: ws})(\texttt{queue:}\ (y|q))\ \ \textbf{if}\ \ l \doteq \texttt{rs},$
- $rw_{\texttt{try}} \triangleq \texttt{try}((y\ ys))(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow$
  $\texttt{try}((y\ ys))(\texttt{pc[}y\texttt{]: cs})(\texttt{queue:}\ q)\ \textbf{if}\ l \doteq \texttt{ws}\ \textbf{and}\ top(q) \doteq y,$
- $rw_{\texttt{exit}} \triangleq \texttt{exit}((y\ ys))(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow$
  $\texttt{exit}((y\ ys))(\texttt{pc[}y\texttt{]: rs})(\texttt{queue: dequeue}(q))\ \textbf{if}\ l \doteq \texttt{cs}$

The sort `SetPid` denotes sets of process identifiers. Given a term ps denoting a set of process identifiers, $\texttt{init}(ps)$ denotes the initial state when the processes participate in Qlock. Three rewriting rules $rw_{\texttt{want}}$, $rw_{\texttt{try}}$ and $rw_{\texttt{exit}}$ in $\mathcal{R}_{\mathrm{Qlock}}$ specify three sets of transitions $\{want_1, want_2, \ldots\}$, $\{try_1, try_2, \ldots\}$, and $\{exit_1, exit_2, \ldots\}$ in $\mathcal{M}_{\mathrm{Qlock}}$.

3 State Machines Specifiable in RWT Specs

In RWT Specs, segments of states used in rewriting rules consist of one action component, and a finite collection of observable components. When a rewriting rule is applied to a state, only a segment of the state that matches the LHS can be changed and the rest keeps unchanged. Since one observable component corresponds to an element in a state in state machines, the number of values that are changed by a rewriting rule must be finite. Hence, if a rewriting rule can be declared for a transition $(u, u') \in t$ in a state machine, the number of data fields in u that are different from their corresponding data fields in u' must

be finite, and the changes of these data fields from u to u' depends on a finite number of data fields in u.

However, the condition is not sufficient. Let us consider a state machine where states are of type $\{l_0 : \mathbb{N}, l_1 : \mathbb{N}, \ldots\}$. The initial state is $\{l_0 = 0, l_1 = 0, l_2 = 0, l_3 = 0, \ldots\}$. There is only one transition *inc* s.t. $(u, u') \in inc$ iff for each $i : \mathbb{N}$ if $i \leq l_0(s)$ then $l_i(u') = l_i(u) + 1$, otherwise, $l_i(u') = l_i(u)$. The transition chain from the initial state is like:

$$\{l_0 = 0, l_1 = 0, l_2 = 0, l_3 = 0, \ldots\} \xrightarrow{inc} \{l_0 = 1, l_1 = 0, l_2 = 0, l_3 = 0, \ldots\} \xrightarrow{inc}$$
$$\{l_0 = 2, l_1 = 1, l_2 = 0, l_3 = 0, \ldots\} \xrightarrow{inc} \{l_0 = 3, l_1 = 2, l_2 = 1, l_3 = 0, \ldots\} \xrightarrow{inc} \ldots$$

The transition *inc* cannot be specified in the way of RWT Specs, because the numbers of changed natural numbers from u to u' vary for all $(u, u') \in inc$, although *inc* can be specified in equational theories. After each transition, the number of changed natural numbers is increased and unbounded. Hence, for each transition $t \in T$ in a state machine which is specifiable in an RWT Spec, there must be only a bounded number of elements changed from u to u' for each $(u, u') \in t$. Moreover, each changed value in u' must depend upon a bounded number of elements in u. We call the state machines that satisfy the condition are *double bounded*, and corresponding EQT Specs *double bounded* EQT Specs. Any state machines that can be specified in RWT Specs are double bounded.

To automatically generate an RWT Spec from an EQT Spec, we first need to check if the EQT Spec, namely the state machine denoted by it, is double bounded. However, it is not decidable for all EQT Specs whether they are double bounded. We have such a concrete EQT Spec which cannot be decided to be double bounded or not.

Let us consider an EQT Spec $\mathcal{S}_{\mathrm{PCP}}$ that specifies a state machine of finding solutions to Post's Correspondence Problem (PCP)[12]. Let `pcp-instance` be an arbitrary instance of PCP on the alphabet $\{a, b\}$, and `Seq` be a sort for sequences of natural numbers.

- $\mathcal{O}_{\mathrm{PCP}} \triangleq \{\mathtt{isSolution} : \Upsilon\ \mathtt{Seq} \rightarrow \mathtt{Bool}\}$
- $\mathcal{T}_{\mathrm{PCP}} \triangleq \{\mathtt{solve} : \Upsilon \rightarrow \Upsilon\}$
- $\mathcal{E}_{\mathrm{PCP}} \triangleq \{\mathcal{E}_{\mathtt{init}}, \mathcal{E}_{\mathtt{solve}}\}$
  - $\mathcal{E}_{\mathtt{init}} \triangleq \{\mathtt{isSolution}(\mathtt{init}, sq) = \mathtt{false}\}$
  - $\mathcal{E}_{\mathtt{solve}} \triangleq \{\mathtt{isSolution}(\mathtt{solve}(\upsilon), sq) = \mathtt{check}(\mathtt{pcp\text{-}instance}, sq)\}$

where, $\mathtt{isSolution}(\upsilon, sq)$ denotes if sq is a solution to `pcp-instance` in υ, and $\mathtt{solve}(\upsilon)$ denotes a successor of υ. The function symbol `check` denotes a function that checks if sq is a solution to `pcp-instance`. Since it is undecidable if `pcp-instance` has solutions according to the undecidability of PCP, it is also undecidable if the number of values observed by `isSolution` and changed from υ to its successor state $\mathtt{solve}(\upsilon)$ is bounded.

To automate the translation from EQT Specs into RWT Specs, some constraints need to be imposed on EQT Specs. We focus on a specific class of double bounded EQT Specs called EADS Specs. Without loss of generality, we suppose that a special sort `Pid` is predefined for the processes (or principals) in

dynamic systems. All EADS Specs must conform to the following syntax-level constraints:

1. Each $o \in \mathcal{O}$ should be declared in the form of $o : \Upsilon \rightarrow D_o$ or $o : \Upsilon\ \mathtt{Pid} \rightarrow D_o$;
2. If there exists $o \in \mathcal{O}$ s.t. $o : \Upsilon\ \mathtt{Pid} \rightarrow D_o$, the declaration of each $a \in \mathcal{A}$ should be one of the following two forms:
   (a) $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \rightarrow \Upsilon$, where each D_{ai} cannot be Pid[3];
   (b) $a : \Upsilon\ \mathtt{Pid}\ \{D_{a2} \dots D_{an}\} \rightarrow \Upsilon$;
   Otherwise, there is no restriction on the declaration of each $a \in \mathcal{A}$;
3. If there exists $o \in \mathcal{O}$ s.t. $o : \Upsilon\ \mathtt{Pid} \rightarrow D_o$, equations declared for t w.r.t o are in one of the following forms:
   (a) for $o : \Upsilon \rightarrow D_o$ and $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \rightarrow \Upsilon$ (D_{ai} cannot be Pid):
   $o(a(\upsilon\{, y_1, \dots, y_n\})) = T_{oa}$ **if** $c\text{-}a(\upsilon\{, y_1, \dots, y_n\})$;
   (b) for $o : \Upsilon\ \mathtt{Pid} \rightarrow D_o$ and $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \rightarrow \Upsilon$ (D_{ai} cannot be Pid):
   $o(a(\upsilon\{, y_1, \dots, y_n\}), y) = o(\upsilon, y)$;
   (c) for $o : \Upsilon \rightarrow D_o$ and $a : \Upsilon\ \mathtt{Pid}\ \{D_{a2} \dots D_{an}\} \rightarrow \Upsilon$:
   $o(a(\upsilon, y_1\{, y_2, \dots, y_n\})) = T_{oa}$ **if** $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$;
   (d) for $o : \Upsilon\ \mathtt{Pid} \rightarrow D_o$ and $a : \Upsilon\ \mathtt{Pid}\ \{D_{a2} \dots D_{an}\} \rightarrow \Upsilon$:
   $o(a(\upsilon, y_1\{, y_2, \dots, y_n\}), y) = (\textbf{if}\ y \doteq y_1\ \textbf{then}\ T_{oa}\ \textbf{else}\ o(\upsilon, y)\ \textbf{fi})\ \textbf{if}$ $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$;

   where, T_{oa} is a term which represents the result into which the value observed by o is changed by action a. If all observers $o \in \mathcal{O}$ are in the form of $o : \Upsilon \rightarrow D_o$, equations must be in form of (3a), but each D_{ai} can be any sort for data elements.
4. All observers $o' \in \mathcal{O}$ (can be o) in T_{oa} and $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$ must be used in the form of $o'(\upsilon\{, y_1\})$;
5. Only observers, actions and the function symbol $c\text{-}a$ associated to each action a can have Υ in their arity;
6. No actions are used in $oa(\upsilon, y_1, \{y_2, \dots, y_n\})$ and $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$.

Assume an EADS Spec specifies a state machine $\mathcal{M}$. Constraint 1 indicates that there are only two kinds of data fields in $\mathcal{M}$. One is called *system-level* data field which is denoted by $o : \Upsilon \rightarrow D_o$ and the other is *process-level* data field denoted by $o : \Upsilon\ \mathtt{Pid} \rightarrow D_o$. Constraint 2 indicates whenever there are process-level data fields in $\mathcal{M}$, only two kinds of transitions are allowed in $\mathcal{M}$. One is called *system-level* transition represented by $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \rightarrow \Upsilon$ and the other is *process-level* transition represented by $a : \Upsilon\ \mathtt{Pid}\ \{D_{a2} \dots D_{an}\} \rightarrow \Upsilon$. Constraint 3 assures that only a bounded number of values in data fields in a state are changed by a transition. Equations 3a and 3b indicate that in the dynamic system only system-level values can be changed by system-level transitions, and equations 3c and 3d indicate a process-level transition can only change system-level values and the process's own values. Constraint 4 indicates a process-level transition executed by a process can only access the system-level data fields and the process-level data fields owned by the process. Constraint 5 guarantees that the number of terms representing data fields in T_{oa} is bounded and the result

[3] Contents in { and } may or may not occur.

of T_{oa} depends on only these terms, namely that the change of each data field depends upon a bounded number of data fields, so does each condition for each action. Constraint 6 assures that each action can be interpreted as a transition.

EADS Specs specify a class of asynchronous distributed systems such as communication protocols and distributed mutual exclusion protocols, and some class of asynchronous shared-memory systems such as Qlock. These systems are characterized by the two main features: (1) A system consists of multiple processes (or principals, etc.) and some shared resources, and (2) Each process (or principal) has only bounded number of components, and each process is only allowed to access and modify its own components, besides shared resources.

4 Translation Strategy

The translation from an EADS Spec to an RWT Spec consists of two phases. The first phase is to construct observable component constructors $\mathcal{OC}$ and action component constructors $\mathcal{AC}$ from $\mathcal{O}$ and $\mathcal{A}$, and to generate rewriting rules $\mathcal{R}$ from $\mathcal{E}$. The second phase includes optimizations of translated RWT Specs and the construction of initial states for the optimized RWT Specs.

4.1 Generation of $\mathcal{OC}$ and $\mathcal{AC}$

Observable component constructors $\mathcal{OC}$ and action component constructors $\mathcal{AC}$ can be directly generated from the declarations of observers $\mathcal{O}$ and actions $\mathcal{A}$. Fig. 2 shows the translation from the declarations of observers and actions to declarations of both observable and action component constructors.

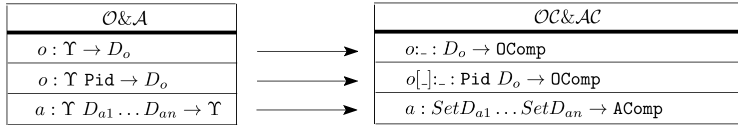


Fig. 2. The translation from $\mathcal{O}$ and $\mathcal{A}$ into $\mathcal{OC}$ and $\mathcal{AC}$

4.2 Generation of $\mathcal{R}$

For each action $a \in \mathcal{A}$ in an EADS Spec $\mathcal{S}$, we construct a rewriting rule to specify the same set of transitions that are denoted by a in a state machine.

For $a \in \mathcal{A}$ s.t. $a : \Upsilon\ D_{a1} \ldots D_{an} \rightarrow \Upsilon (n \geq 0)$, a denotes a set of system-level transitions, by which only system-level data fields can be accessed. Therefore, we only need to consider those observers that denote system-level data fields in a state υ. For $o \in \mathcal{O}$ s.t. $o : \Upsilon \rightarrow D_o$, $o(\upsilon)$ denotes the value of a system-level data field in υ. According to the equation 3a, $o(\upsilon)$ is changed into $T_{oa\downarrow\mathcal{S}}$[4] when c-$a(\upsilon)$ holds. We introduce a fresh variable d_o of D_o denoting the value denoted by $o(\upsilon)$ in a data field. We assume there are $m (m \geq 1)$ observers s.t. $\mathcal{O} = \{o_1, \ldots, o_m\}$. A rewriting rule that is constructed from a and $\mathcal{E}_a$ is as follow:

[4] $T_{oa\downarrow\mathcal{S}}$ represents the canonical form of T_{oa} in context $\mathcal{S}$.

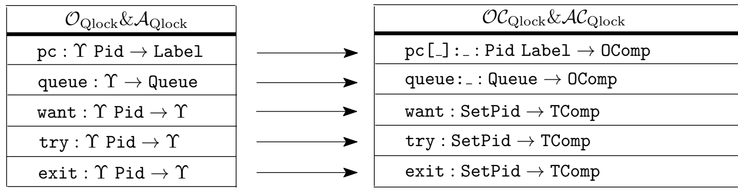


Fig. 3. The translation from $\mathcal{O}_{\text{Qlock}}$ and $\mathcal{A}_{\text{Qlock}}$ into $\mathcal{OC}_{\text{Qlock}}$ and $\mathcal{AC}_{\text{Qlock}}$

$a((y_1\ ys_1), \ldots, (y_n, ys_n))(o_1\colon o_1(\upsilon)) \ldots (o_m\colon o_m(\upsilon)) \Rightarrow a(y_1\ ys_1), \ldots, (y_n, ys_n))$
$(o_1\colon T_{o_1 a \downarrow \mathcal{S}}) \ldots (o_m\colon T_{o_m a \downarrow \mathcal{S}})$ **if** $c\text{-}t(\upsilon)_{\downarrow \mathcal{S}}$,

with terms $o_1(\upsilon), \ldots, o_m(\upsilon)$ substituted by $d_{o_1}, \ldots, d_{o_m}$ respectively. In the rewriting rule, each component at LHS has a corresponding successor at RHS. The action component (if there is) keeps unchanged. The change of observable components like from $(o : o(\upsilon))$ to $(o\colon T_{oa\downarrow\mathcal{S}})$ exactly denotes the one from $o(\upsilon)$ to $T_{oa\downarrow\mathcal{S}}$ in the original EQT Spec. Note that $T_{oa\downarrow\mathcal{S}}$ can be $o(\upsilon)$, which means that corresponding shared resource is not changed. If $o(\upsilon)$ is also not used by other observable component, it can be removed from the rewriting rule. This step is called *optimization* (see Subsection 4.3 for details). Moreover, if all observers $o \in \mathcal{O}$ are declared like $o : \Upsilon \rightarrow D_o$, D_{ai} in the declaration of a can be any sort, otherwise, D_{ai} cannot be `Pid`, according to Constraint 2.

For $a \in \mathcal{A}$ s.t. $a : \Upsilon$ `Pid` $D_{a2} \ldots D_{an} \rightarrow \Upsilon (n \geq 0)$, a denotes a process-level transition. Besides system-level data fields, a process-level transition can access process-level data fields in the process where the transition takes place. Let y_1 be a variable of `Pid` and $y_2, \ldots, y_n$ be variables of $D_{a2}, \ldots, D_{an}$ respectively. We consider a process-level transition denoted by a w.r.t. y_1 and parameters $y_2, \ldots, y_n$. For each $o \in \mathcal{O}$ s.t. $o : \Upsilon \rightarrow D_o$, we deal with it similarly like in the construction of rewriting rules for system-level transitions. For each $o \in \mathcal{O}$ s.t. $o : \Upsilon$ `Pid` $\rightarrow D_o$, among process-level data fields denoted by o, only those owned by y_1 can be accessed. In the state denoted by υ, the value in a process-level data field of the process y_1 w.r.t. o is denoted by $o(\upsilon, y_1)$. According to the equation 3d, it is changed into $T_{oa\downarrow\mathcal{S}}$ in the successor $a(\upsilon, y_1, y_2, \ldots, y_n)$ under the condition that $c\text{-}a(\upsilon, y_1, y_2, \ldots, y_n)$ holds. We introduce a fresh variable d_o of D_o corresponding to $o(\upsilon, y_1)$. We assume that the first k observers are in the form of $o : \Upsilon \rightarrow D_o$ and rest of $o : \Upsilon$ `Pid` $\rightarrow D_o$ in m observers $\{o_1, \ldots, o_k, o_{k+1}, \ldots, o_m\}$. A rewriting rule specifying a set of system-level transitions denoted by action a and $\mathcal{E}_a$ w.r.t. $y_1, y_2, \ldots, y_n$ is as follow:

$a((y_1\ ys_1), (y_2\ ys_2), \ldots, (y_n\ ys_n))\ (o_1\colon\ o_1(\upsilon))\ (o_k\colon\ o_k(\upsilon))(o_{k+1}[y_1]\colon$
$o_{k+1}(\upsilon, y_1)) \ldots (o_m[y_1]\colon\ o_m(\upsilon, y_1)) \Rightarrow a((y_1\ ps_1), (y_2\ ys_2), \ldots, (y_n\ ys_n))$
$(o_1\colon\ T_{o_1 a\downarrow\mathcal{S}}) \ldots (o_k\colon\ T_{o_k a\downarrow\mathcal{S}})(o_{k+1}[y_1]\colon\ T_{o_{k+1} a\downarrow\mathcal{S}}) \ldots (o_m[y_1]\colon\ T_{o_m a\downarrow\mathcal{S}})$
if $c\text{-}t(\upsilon, y_1, y_2, \ldots, y_n)_{\downarrow\mathcal{S}}$,

with terms $o_1(\upsilon), \ldots, o_k(\upsilon), o_{k+1}(\upsilon, y_1), \ldots, o_m(\upsilon, y_1)$ substituted by variables $d_{o_1}, \ldots, d_{o_k}, d_{o_{k+1}}, \ldots, d_{o_m}$, respectively.

For instance, $\mathcal{S}_{\text{Qlock}}$ is an EADS Spec, according to the four restrictions. Fig. 3 shows the translation of the declarations of observers and actions in $\mathcal{S}_{\text{Qlock}}$ into the declarations of observable component constructors and action component constructors in $\mathfrak{S}_{\text{Qlock}}$. According to $\mathcal{E}_{\texttt{want}}$ in $\mathcal{S}_{\text{Qlock}}$, we construct the following rewriting rule to specify the set of transitions denoted by the action `want`:

$\texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ \texttt{pc}(\upsilon, y))\ (\texttt{queue:}\ \texttt{queue}(\upsilon)) \Rightarrow \texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}$
$\texttt{pc}(\texttt{want}(\upsilon, y), y)_{\downarrow\mathcal{S}_{\text{Qlock}}})\ (\texttt{queue:}\ \texttt{queue}(\texttt{want}(\upsilon, y))_{\downarrow\mathcal{S}_{\text{Qlock}}}\ \textbf{if}\ \texttt{c-want}(\upsilon, y)_{\downarrow\mathcal{S}_{\text{Qlock}}}.$

According to $\mathcal{E}_{\texttt{want}}$, $\texttt{pc}(\texttt{want}(\upsilon, y), y)$ is reduced to ws, $queue(\texttt{want}(\upsilon, y))$ to $(y|queue(\upsilon))$ and $\texttt{c-want}(\upsilon, y)$ to $\texttt{pc}(\upsilon, y) \doteq \text{rs}$. Consequently, we obtain the following rewriting rule:

$\texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ pc(\upsilon, y))\ (\texttt{queue:}\ queue(\upsilon)) \Rightarrow \texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ \texttt{ws})$
$(\texttt{queue:}\ (q|queue(\upsilon)))\ \textbf{if}\ \texttt{pc}(\upsilon, y) \doteq \texttt{rs}.$

Further, we substitute l for $pc(\upsilon, y)$ and q for $queue(\upsilon)$, then we obtain the rewriting rule $rw_{\texttt{want}}$:

$\texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ l)\ (\texttt{queue:}\ q) \Rightarrow \texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ \texttt{ws})\ (\texttt{queue:}\ (y|q))$
$\textbf{if}\ l \doteq \texttt{rs}.$

Similarly, we can construct the rewriting rules $rw_{\texttt{try}}$ and $rw_{\texttt{exit}}$ for the actions `try` and `exit` in $\mathcal{S}_{\text{Qlock}}$.

4.3 Optimization of RWT Specs

Generated RWT Specs need to be optimized so that they can be efficiently model checked in Maude. In Maude, rewriting with both equations and rules takes place by matching an LHS against a subject term and evaluating the corresponding condition [5, chap. 1]. Hence, the less complex the LHS and the condition of a rewriting rule are, the less time it takes to match a term to the LHS and to evaluate the condition, respectively.

A general way of optimizing rewriting rules is deleting redundant terms. In an RWT Spec, action components are not changed in rewriting rules. From the program point of view, it provides necessary variables that guarantee the rewriting rule is executable, because Maude generally requires variables that occur in the RHS or condition must occur in the LHS to make rewriting rules executable [5, chap. 6]. However, some variables in an action component may be also used by some observable components at the LHS in rewriting rules. In this situation, these variables in the action component become redundant. We take the rewriting rule $rw_{\texttt{want}}$ in $\mathfrak{S}_{\text{Qlock}}$ for instance. The variable y in $\texttt{want}((y\ ys))$ is also used in $(\texttt{pc}[y]\texttt{:}\ l)$. Since there is only one parameter taken by the action component constructor *want*, deleting x means that we can delete the whole action component. After deleting $\texttt{want}((y\ ys))$, we obtain a simpler rewriting rule, as follow:

$$(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow (\texttt{pc[}y\texttt{]:}\ \texttt{ws})(\texttt{queue:}\ (y|q))\ \textbf{if}\ l \doteq \texttt{rs}.$$

Another case is that when a parameter $y_k, k \in \{1, \ldots, n\}$ in an action component of a occurs in some observable components at LHS of a rewriting rule or y_k occurs neither in any observable components at RHS nor in condition, we can

remove the kth parameter of a, and consequently revise the declaration of a in $\mathcal{AC}$. If all parameters of a are removed, the action component can be removed.

Redundant observable components in rewriting rules can also be deleted. An observable component is redundant when the value in it is neither changed by the transition, nor used by other components or in conditions. A redundant observable component can be deleted directly from both the sides of rewriting rules, without changing the meaning of the rewriting rules.

Another optimization is to simplify or delete the condition of a rewriting rule. The optimization is achieved by *equivalent replacement*. We assume that the condition is a conjunction. If a conjunct in the condition is an equivalence relation in the form of $x \doteq T$ and x occurs in neither T nor the other part of the condition, where x is a variable and T is a term, we can replace x that occurs in the both sides of the rewriting rule with T and delete the conjunct from the condition. For instance, the rewriting rule can be further simplified to be the following one:

$$(\texttt{pc[}y\texttt{]: rs})(\texttt{queue: } q) \Rightarrow (\texttt{pc[}y\texttt{]: ws})(\texttt{queue: } (q|y)).$$

An optimized RWT Spec of Qlock is as follow:

$\mathfrak{S}'_{\text{Qlock}}$

$\mathcal{OC}'_{\text{Qlock}} \triangleq \{\texttt{pc[_]:_: Pid Label} \rightarrow \texttt{OComp}, \texttt{queue:_: Queue} \rightarrow \texttt{OComp}\};$
$\mathcal{AC}'_{\text{Qlock}} \triangleq \emptyset;$
$\mathcal{F}'_{\text{Qlock}} \triangleq \{\texttt{init: SetPid} \rightarrow \texttt{State}, \texttt{mk-pc: SetPid} \rightarrow \texttt{State}\};$
$\mathcal{E}'_{\mathcal{F}'_{\text{Qlock}}} \triangleq \{\texttt{init}(ys) = (\texttt{queue: empty})\ \texttt{mk-pc}(ys),$
$\quad \texttt{mk-pc}(\texttt{empty-set}) = \texttt{empty-state},$
$\quad \texttt{mk-pc}(y\ ys) = (\texttt{pc[}y\texttt{]: rs})\ \texttt{mk-pc}(ys).\}$
$\mathcal{R}'_{\text{Qlock}} \triangleq \{rw_{\texttt{want}}, rw_{\texttt{try}}, rw_{\texttt{exit}}\}$, where:

- $rw_{\texttt{want}} \triangleq (\texttt{pc[}y\texttt{]: rs})(\texttt{queue: } q) \Rightarrow (\texttt{pc[}y\texttt{]: ws})(\texttt{queue: } (y|q));$
- $rw_{\texttt{try}} \triangleq (\texttt{pc[}y\texttt{]: ws})(\texttt{queue: } q) \Rightarrow (\texttt{pc[}y\texttt{]: cs})(\texttt{queue: } q)$ **if** $\texttt{top}(q) \doteq y;$
- $rw_{\texttt{exit}} \triangleq (\texttt{pc[}y\texttt{]: cs})(\texttt{queue: } q) \Rightarrow (\texttt{pc[}y\texttt{]: rs})(\texttt{queue: dequeue}(q)).$

4.4 Generation of $\mathcal{F}$ and $\mathcal{E}_{\mathcal{F}}$

The last step is to construct a set $\mathcal{F}$ of function symbols and a set of equations $\mathcal{E}_{\mathcal{F}}$ for $\mathcal{F}$ to specify initial states in an RWT Spec $\mathfrak{S}$, according to the specification of the initial states denoted by *init* in the original EQT Spec $\mathcal{S}$.

For each $o \in \mathcal{O}$ in $\mathcal{S}$ s.t. $o : \Upsilon \rightarrow D_o$, the value of the data field corresponding to o in initial states is $o(init)_{\downarrow \mathcal{S}}$. Consequently, an observable component $(o : o(init)_{\downarrow \mathcal{S}})$ in $\mathfrak{S}$ can be constructed to correspond to the data field. For each observer $o \in \mathcal{O}$ s.t. $o : \Upsilon\ \texttt{Pid} \rightarrow D_o$, the value of the data field corresponding to o with a process y in initial states is denoted by $o(init, y)_{\downarrow \mathcal{S}}$. Hence, the data field can be denoted by the observable component $(o[y] : o(init, y)_{\downarrow \mathcal{S}})$. Given a set of processes, to construct a set of observable components that represent all data fields corresponding to o in the initial states, we define an auxiliary function which is denoted by $mk\text{-}o : \texttt{PidSet} \rightarrow \texttt{State}$. The following two equations are declared for $mk\text{-}o$:

$mk\text{-}o(\texttt{empty-set}) = \texttt{empty-state}$,
$mk\text{-}o(y\ ys) = (o[y] : o(init, y)_{\downarrow\mathcal{S}})\ make\text{-}o(ys)$,

where y is a variable of `Pid` and ys of `SetPid`. Let $sa_i (1 \leq i \leq n)$ denote a set of elements of $SetD_{ai}$ that are used in the specified system. The action component of a in the initial states can be constructed as $a(sa_1, \ldots, sa_n)$, for each $a : SetD_{a1} \ldots SetD_{an} \rightarrow \texttt{TComp}$.

We suppose $\mathcal{OC}$ consists of m observable component constructors, where the first k constructors are declared as o_i:_ $: D_{oi} \rightarrow \texttt{OComp}$ for $i = 1, \ldots, k$, and the rest as o_i[_]:_ $: \texttt{Pid}\ D_{oi} \rightarrow \texttt{OComp}$ for $i = k+1, \ldots, m$. We also suppose $\mathcal{OA}$ consists of n action component constructors $a_1, \ldots, a_n$ and each a_j takes l_j parameters for $j = 1, \ldots, n$. Let $\{SetD_1, \ldots, SetD_{n'}\}$ be the set of all sorts that are taken by at least one component constructor in $\mathcal{AC}$. We declare a function symbol `init` s.t. $\texttt{init} : \texttt{SetPid}\ SetD_1 \ldots SetD_{n'} \rightarrow \texttt{State}$, and declare the following equation for `init`:

$\texttt{init}(ys, sd_1, \ldots, sd_{n'}) = (o_1 : o_1(init)_{\downarrow\mathcal{S}}) \ldots (o_k : o_k(init)_{\downarrow\mathcal{S}})\ mk\text{-}o_{k+1}(ys) \ldots$
$mk\text{-}o_m(ys)\ a_1(sd1_1, \ldots, sd1_{l_1}) \ldots a_n(sdn_1, \ldots, sdn_{l_n})$,

where each $sd_i (1 \leq i \leq n')$ is a variable of $SetD_i$, and each $sdj_w (1 \leq j \leq n, 1 \leq w \leq l_j)$ is one of $sd_1, \ldots, sd_{n'}$. Consequently, we obtain a set of function symbols $\mathcal{F} = \{\texttt{init}, mk\text{-}o_{k+1}, \ldots, mk\text{-}o_m\}$, and a set $\mathcal{E}_\mathcal{F}$ of equations that are declared for `init` and $mk\text{-}o_{k+1}, \ldots, mk\text{-}o_n$.

We take the construction of $\mathcal{F}'_{\text{Qlock}}$ and $\mathcal{E}'_{\mathcal{F}'_{\text{Qlock}}}$ for $\mathfrak{S}'_{\text{Qlock}}$ as an example. Since $\mathcal{AC}'_{\text{Qlock}}$ is empty, we only need to consider $\mathcal{OC}'_{\text{Qlock}}$. Sort `Pid` is taken as a parameter sort, therefore `init` is declared as $\texttt{init} : \texttt{PidSet} \rightarrow \texttt{State}$, and we have $\texttt{init}(ys) = (\texttt{queue:}\ queue(init)_{\downarrow\mathcal{S}_{\text{Qlock}}})\ \texttt{mk-pc}(ys)$. That is $\texttt{init}(ys) = (\texttt{queue:empty})\ \texttt{mk-pc}(ys)$, and the equations declared for $\texttt{mk-pc} : \texttt{PidSet} \rightarrow \texttt{State}$ are as follows:

$\texttt{mk-pc}(\texttt{empty-set}) = \texttt{empty-state}$
$\texttt{mk-pc}(y\ ys) = (\texttt{pc}[y]\texttt{:}\ \texttt{rs})\ \texttt{mk-pc}(ys)$.

4.5 Principles of Defining EADS Specs for Efficient RWT Specs

Given an extended asynchronous distributed system, we can develop one or more EADS Specs, which consequently correspond to different RWT Specs generated by the proposed strategy. The efficiency of RWT Specs varies according to the complexity of rewriting rules. Some principles should be followed to develop EADS Specs from which efficient RWT Specs can be generated: (1) condition should be in conjunction form if possible, and (2) each conjunct should be in the form of $o(\upsilon\{, x_1\}) \doteq T$ if possible, where T is a term.

We take $rw_{\texttt{try}}$ in $\mathfrak{S}_{\text{Qlock}}$ for instance. To be able to remove the condition $\texttt{top}(q) \doteq y$, we need to revise the condition $\texttt{c-try}(\upsilon, y)$ in $\mathcal{S}_{\text{Qlock}}$, according to the two principles. The condition $\texttt{top}(q) \doteq y$ corresponds to $\texttt{top}(\texttt{queue}(\upsilon)) = y$ in $\mathcal{S}_{\text{Qlock}}$ which means that y is at the top of the shared queue in state υ. An equivalent condition is $\texttt{queue}(\upsilon) \doteq (y \mid q)$. Consequently, we need to revise the declaration of try and $\mathcal{E}_{\texttt{try}}$, like:

$try : \Upsilon$ Pid Queue $\rightarrow \Upsilon$, and
$\mathcal{E}_{\texttt{try}} \triangleq \{\texttt{c-try}(\upsilon, y, q) = \texttt{pc}(\upsilon, y) \doteq \texttt{ws} \wedge \texttt{queue}(\upsilon) \doteq (y \mid q),$
$\texttt{pc}(\texttt{try}(\upsilon, y, q), x) = \textbf{if } x \doteq y \textbf{ then } \texttt{cs} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi if } \texttt{c-try}(\upsilon, y, q),$
$\texttt{queue}(\texttt{try}(\upsilon, y, q)) = \texttt{queue}(\upsilon) \textbf{ if } \texttt{c-try}(\upsilon, y, q),$
$\texttt{try}(\upsilon, y, q) = \upsilon \textbf{ if } \neg\texttt{c-try}(\upsilon, y, q)\}$

The translated rewriting rule of the modified $\mathcal{E}_{\texttt{try}}$ is:

$\texttt{try}((y\ ys), (q_1\ qs))\texttt{(pc[}y\texttt{]: }l\texttt{)(queue: }q\texttt{)} \Rightarrow \texttt{try}((y\ ys), (q_1\ qs))$
$\texttt{(pc[}y\texttt{]: cs)(queue: }q\texttt{)} \textbf{ if } l \doteq \texttt{ws} \textbf{ and } q \doteq (y \mid q_1),$

where q_1 is a variable of Queue and qs_1 of `SetQueue`. After the optimization, we obtain a much simpler rewriting rule $rw_{\texttt{try}}$:

$\texttt{(pc[}y\texttt{]: ws)(queue: }(y \mid q_1)\texttt{)} \Rightarrow \texttt{(pc[}y\texttt{]: cs)(queue: }(y \mid q_1)\texttt{)}$

5 Experimental Results

To the best of our knowledge, three strategies for the translation of specifications of state machines from CafeOBJ to Maude have been proposed. An implementation of the most straightforward strategy (called TS1) is described in [11]. Another strategy (called TS2) is proposed and its implementation is described in [9]. Yet another strategy (called TS3) is proposed in [10] and its implementation is described in [11]. We take Qlock and NSPK as examples to evaluate the efficiency of the specifications translated by our proposed strategy by comparing with the three strategies. The efficiency is measured by the number of states in the state space from initial states with the same depth and the time that Maude takes to finish searching these states. We use $\mathcal{S}^{\dagger}$, $\mathcal{S}^{\ddagger}$, $\mathcal{S}^{\star}$ and $\mathcal{S}^{\circledast}$ to denote specifications generated by TS1, TS2, TS3 and the proposed one in this paper. $\mathcal{S}^{*}$ denotes manually developed specifications. The experiment has been conducted on Ubuntu in a laptop with 2x1.20GHz Duo Core processor and 4GB memory.

Table 1 shows that the number of states increases with the increase of depth, which consequently causes time on model checking to increase. Except in $\mathcal{S}^{\dagger}_{\text{Qlock}}$, the number of states in other specifications is the same with the same depth. This is because states in $\mathcal{S}^{\dagger}_{\text{Qlock}}$ are implicitly represented like in EQT Specs, while states in other specifications are explicitly represented by a set of components. The time spent in $\mathcal{S}^{\circledast}_{\text{Qlock}}$ is the least and is the same as the one in $\mathcal{S}^{*}_{\text{Qlock}}$, indicating that the efficiency of $\mathcal{S}^{\circledast}_{\text{Qlock}}$ is higher than other translated ones, and equal to the manually developed one. Searching fails when the depth is 9 in both $\mathcal{S}^{\ddagger}_{\text{Qlock}}$ and $\mathcal{S}^{\star}_{\text{Qlock}}$ in a reasonable time, but successfully finishes in $\mathcal{S}^{\circledast}_{\text{Qlock}}$, although the number of the states in the three specifications are the same. This is because the efficiency of a specification depends upon not only the state space, but the forms of rewriting rules in the specification, as explained in Section 4.3.

NSPK is a security protocol to achieve mutual authentication between two principals over network [13]. To generate a rewrite theory specification from the corresponding equational theory specification of NSPK, the existing translation strategies require a fixed number of nonces and messages. Hence, we need to fix

Table 1. Times (ms) taken by Maude for checking the mutual exclusion property of Qlock with 9 processes, and the number of states traversed in different depths

(a) *Time on model checking*

	$\mathcal{S}^{\dagger}_{\text{Qlock}}$	$\mathcal{S}^{\ddagger}_{\text{Qlock}}$	$\mathcal{S}^{\star}_{\text{Qlock}}$	$\mathcal{S}^{\bigast}_{\text{Qlock}}$	$\mathcal{S}^{*}_{\text{Qlock}}$
1	0	0	0	**0**	0
3	116	64	20	**8**	8
5	20361	2596	764	**268**	268
7	–	33458	16373	**5336**	5336
9	–	–	–	**42072**	42072

(b) *The number of states traversed*

	$\mathcal{S}^{\dagger}_{\text{Qlock}}$	$\mathcal{S}^{\ddagger}_{\text{Qlock}}$	$\mathcal{S}^{\star}_{\text{Qlock}}$	$\mathcal{S}^{\bigast}_{\text{Qlock}}$	$\mathcal{S}^{*}_{\text{Qlock}}$
1	10	10	10	**10**	10
3	748	667	667	**667**	667
5	36262	22339	22339	**22339**	22339
7	–	339859	339859	**339859**	339859
9	–	–	–	**1609939**	1609939

both of the numbers of random numbers and principals. However, the number of messages are huge, which consequently makes the terms representing states huge. For instance, 3 principals and 2 random numbers lead to 18 nonces and 32076 different messages. Moreover, the huge number of messages drastically increases the number of rewriting rules in the target specifications, and hence it becomes impossible to reasonably model check the generated specifications. In our approach, we do not need to fix the number of nonces and messages thanks to the optimization, to generate an RWT Spec of NSPK. The generated Maude specification is denoted by $\mathcal{S}^{\bigast}_{\text{NSPK}}$ and the manually developed one by $\mathcal{S}^{*}_{\text{NSPK}}$.

Table 2 shows that the time that is spent on searching in $\mathcal{S}^{\bigast}_{\text{NSPK}}$ is more than in $\mathcal{S}^{*}_{\text{NSPK}}$, although the number of states in $\mathcal{S}^{\bigast}_{\text{NSPK}}$ is the same as the one in $\mathcal{S}^{*}_{\text{NSPK}}$ with the same depth. It indicates a translated specification by the proposed strategy may not be the most optimized one because some optimizations cannot be automatically done. For example, if a condition in a rewriting rule is in the form of $\neg(x \doteq u)$, we cannot automatically transform the rewriting rule into an unconditional one. It takes some more time on pattern matching. However, both of the specifications are comparably efficiently model checked.

Table 2. Times (ms) taken by Maude to model check the secrecy property for NSPK with 3 principals, and the number of states traversed in different depths

(a) *Time on model checking*

	1	2	3	4	5
$\mathcal{S}^{*}_{\text{NSPK}}$	0	4	36	716	24617
$\mathcal{S}^{\bigast}_{\text{NSPK}}$	**0**	**4**	**44**	**1104**	**47654**

(b) *The number of states traversed*

	1	2	3	4	5
$\mathcal{S}^{*}_{\text{NSPK}}$	7	105	1745	33901	710899
$\mathcal{S}^{\bigast}_{\text{NSPK}}$	**7**	**105**	**1745**	**33901**	**710899**

6 Conclusion and Future Work

We have proposed a specific class of EQT Specs called EADS Specs, and proposed a strategy for the translation from EADS Specs into RWT Specs. Case studies have been conducted to show the efficiency of translated specifications is significantly improved. Although the strategy can only deal with EADS Specs, most of EQT Specs that are developed for practical verifications belong to this class based on our experience of theorem proving in CafeOBJ.

Regarding the correctness of the translation, we argue that a counterexample in the generated RWT Spec by the strategy $\mathfrak{S}$ of an EADS Spec $\mathcal{S}$ is also a counterexample in $\mathcal{S}$. We can prove it by show that for any state transition chain in $\mathfrak{S}$, there is a corresponding chain in $\mathcal{S}$. First, we show that for an arbitrary initial state denoted by t_0 in $\mathfrak{S}$, there exists a term (which is actually `init`) in $\mathcal{S}$, corresponding to t_0. Then we assume two arbitrary states denoted by t_i and t_{i+1} in $\mathfrak{S}$ and an arbitrary state denoted by v_i in $\mathcal{S}$ such that t_{i+1} denotes a successor state of the one by t_i and v_i corresponds to t_i. We can show there exists a state denoted by v_{i+1} in $\mathcal{S}$ such that v_{i+1} corresponds to t_{i+1}, and v_{i+1} denotes a successor state of v_i. Hence, we can claim that $\mathfrak{S}$ simulates $\mathcal{S}$, which indicating the correctness of the proposed translation strategy. A detailed proof in theory for the correctness of the translation is one piece of our future work. Moreover, A prototype of the present translation strategy has been implemented and successfully applied to Qlock and NSPK. We will further improve the translator, especially the optimization part, for practical applications.

References

1. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 480–500. Springer, Heidelberg (2007)
2. Eshuis, R.: Translating safe Petri nets to statecharts in a structure-preserving way. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 239–255. Springer, Heidelberg (2009)
3. Vargas, P., et al.: Model Checking LTL Formulae in RAISE with FDR. In: Leuschel, M., Wehrheim, H. (eds.) IFM 2009. LNCS, vol. 5423, pp. 231–245. Springer, Heidelberg (2009)
4. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. World Scientific, Singapore (1998)
5. Clavel, M., Durán, F., et al.: All about Maude. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical foundations and methodologies. Computing and Informatics 22, 257–283 (2003)
7. Holzmann, G.: The SPIN model checker. Addison-Wesley, Reading (2004)
8. 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)
9. Kong, W., Ogata, K., et al.: A Lightweight Integration of Theorem Proving and Model Checking for System Verification. In: 12th APSEC, pp. 59–66 (2005)
10. Nakamura, M., Kong, W., et al.: A Specification Translation from Behavioral Specifications to Rewrite Specifications. IEICE Transactions 91-D, 1492–1503 (2008)
11. Zhang, M., Ogata, K.: Modular implementation of a translator from behavioral specifications to rewrite theory specifications. In: 9th QSIC, pp. 406–411 (2009)
12. Sipser, M.: Introduction to the Theory of Computation. PWS Pub. Co. (1996)
13. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. CACM 21, 993–999 (1978)

Alternating Interval Based Temporal Logics*

Cong Tian and Zhenhua Duan**

Institute of Computing Theory and Technology and ISN Laboratory
Xidian University, Xi'an, 710071, P.R. China
{ctian,zhhduan}@mail.xidian.edu.cn

Abstract. To specify properties of open systems with interval based temporal logics, alternating interval based temporal logics are proposed by introducing Concurrent Game Structures (CGS) to Propositional Projection Temporal Logic (PPTL) and Propositional Interval Temporal Logic (PITL). Further, examples are given to show how properties of open systems can be specified by APTL and AITL formulas. Moreover, to establish the automata based model theory for the new proposed logics, Generalized alternating Büchi automata over Concurrent Game structures (GBCGs) are defined. And a transformation from APTL formulas to GBCGs is presented. In addition, a decision procedure for checking the satisfiability of APTL formulas, and a model checking approach for APTL with Concurrent Game Structures (CGSs) models are presented.

Keywords: Temporal Logic; Open Systems; Model Checking; Concurrent Game Structures; Automata.

1 Introduction

With the development of Internet, open systems [7] have pervaded into our daily life. For a closed system, the behavior is completely determined by its internal components, and cannot be influenced by the environment. In contrast, for an open system, the behavior is jointly determined by the internal components, and the inputs received from the environment. The widely used distributed systems such as composite Web services and P2P networks, as well as embedded systems and control systems are all open systems.

To verify safety critical systems, the traditional closed systems are often modeled in terms of transition systems or Kripke structures consisting of a set of system states and a set of transitions among the states. For open systems, transition systems and Kripke structures are generalized by games to express the interaction between the system and its environment, or the interaction among several different agents or players that work concurrently [8]. Practically, two-player (or agent) games are often used to model the interaction between the system and its environment, while n-player games are convenient for expressing the interaction among different concurrent components. As a result, Concurrent Game Structure (CGS) [1] is one of the most frequently used models for the modeling and verifying of open systems.

* This research is supported by the NSFC Grant No. 61003078, 60433010, 60873018 and 60910004, National Program on Key Basic Research Project of China (973 Program) Grant No.2010CB328102 and SRFDP Grant 200807010012.

** Corresponding author.

J.S. Dong and H. Zhu (Eds.): ICFEM 2010, LNCS 6447, pp. 694–709, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Primitively, for model checking [6] purpose of open systems, temporal logic syntaxes developed for specifying closed systems are applied to reformulate the new semantical conditions of open systems. Due to this, alternating properties of interactions among different concurrent agents cannot be well captured. To overcome this disadvantage, enriched temporal logics [1] such as Alternating Temporal Logic (ATL), ATL* and Alternating Mu-Calculus (AMC) are introduced by extending Computing Tree Logic (CTL), CTL* [9,10] and Mu-calculus [11] with the set of agents respectively. In stead of transition systems and Kripke structures, these logics are interpreted over concurrent game structures. Further, to capture composite properties of open systems, instead of two-player games between the system and its environment, the more general setting of n-player games, with a set of players that represent different concurrent agents are considered.

Within the community of temporal logics, interval based temporal logics (IBTL) simplify the formulation of certain correctness properties [12], and are useful in specification and verification of concurrent systems. Among the interval based temporal logics, Interval Temporal Logic (ITL) [13] and Projection Temporal Logic (PTL) [14] have been widely investigated and several tools have been developed for supporting the verification with ITL and PTL [4,16]. Compared to LTL, CTL, CTL* and Mu-Calculus, ITL and PTL have several merits: (1) Propositional PTL (PPTL) is more powerful since it has the expressiveness of full regular expressions [17]. In contrast, PLTL, CTL and CTL* are less expressive than full regular expressions. Even though Mu-calculus has the same expressive power as PITL and PPTL, however, it is inconvenient to specify properties with Mu-calculus. (2) Intervals are convenient to specify state sensitive properties. For example, to specify "p holds between the 10th and 16th states, a formula $len(10); len(6) \wedge \Diamond p; true$ in PITL and PPTL works. However, it is somewhat hard to specify this property in PLTL and any other temporal logics. (3) Chop and projection constructs are useful for specifying properties of sequential and iterative behaviors respectively. And these properties cannot (or with difficulty) be described by LTL, CTL, CTL* or Mu-Calculus.

With this motivation, we extend PPTL and PITL to Alternating Projection Temporal Logic (APTL) and Alternating Interval Temporal Logic (AITL). Further, as an example, a train packing problem is employed to show how properties of open systems can be specified by APTL and AITL formulas. Moreover, to establish the automata based model theory for the new proposed logics, Generalized alternating Büchi automata over Concurrent Game structures (GBCGs) are defined. Moreover, a transformation from APTL formulas to GBCGs is presented. To make the transformation efficient, Normal Forms (NFs) and Complete Normal Forms (CNFs) are defined; further, Normal Form Graphs (NFGs) and Labeled NFGs (LNFGs) are constructed for APTL formulas. Accordingly, an automata based approach for checking the satisfiability of APTL formulas and model checking APTL are obtained.

The paper is organized as follows. The next section briefly presents CGSs and automata over CGSs. In Section 3, the syntax and semantics of APTL and AITL are given; further, how properties can be specified by these logics is illustrated. In Section 4, normal forms and complete normal forms as well as NFGs and LNFGs are defined for APTL formulas; accordingly, a transformation from APTL formulas to GBCGs is

achieved. In Section 5, an approach for checking the satisfiability of APTL formulas and an algorithm for model checking APTL are obtained. Finally, conclusions are drawn in Section 6.

2 Concurrent Game Structures

2.1 Concurrent Game Structures

Concurrent game structures (CGS) [1,18,19] generalize labeled transition systems (or pointed Kripke structures) with a set of agents. Here we generalize the notation of CGS for using interval based temporal logics.

Definition 1. A Concurrent Game Structure (CGS) is defined by $C = (\mathcal{P}, \mathcal{A}, S, s_0, l, \Delta, \tau)$, where

- $\mathcal{P}$ is a finite nonempty set of atomic propositions;
- $\mathcal{A}$ is a finite set of agents;
- S is a finite nonempty set of states, with a designated initial state $s_0 \in S$;
- $l : S \to 2^{\mathcal{P}}$ is a labeling function that decorates each state with a subset of the atomic propositions;
- $\Delta^a(s)$ is a nonempty set of possible decisions for an agent $a \in \mathcal{A}$ at state s; $\Delta^A(s) = \Delta^{a_1}(s) \times ... \times \Delta^{a_k}(s)$ is a nonempty set of decision vectors for the set of agents $A = \{a_1, ..., a_k\} \in 2^{\mathcal{A}}$ at state s; accordingly, $\Delta^{\mathcal{A}}(s)$ simplified as $\Delta(s)$ denotes the decisions of all agents in $\mathcal{A}$; and for a decision $d \in \Delta(s)$, d_a denotes the decision of agent a within d, and d_A denotes the decision of the set of agents $A \subseteq \mathcal{A}$ within d;
- For each state $s \in S$, and $d \in \Delta(s)$, a state $\tau(s, d) \in S$ maps s and a decision d of the agents in $\mathcal{A}$ to a new state in S. Note that in a CGS, for a state s, each transition is made by the decision, $d \in \Delta(s)$, of all agents in $\mathcal{A}$. In some cases, if we just concern with the decisions of $A \subseteq \mathcal{A}$ without caring about the ones of other agents, notation d_A is used, particularly, if A is a singleton, d_a is adopted. □

For two states s and s' in S, we say s' is a successor of s if $s' = \tau(s, d)$ with $d \in \Delta(s)$. Thus, s' is a successor of s iff whenever the game is in state s, the agents in $\mathcal{A}$ can make decisions so that s' is the next state. Note that here the concurrent games structures are not total. That is there may exist some state s without any successors. For convenience, a state s without any successors is called a dead state denoted by $\tau(s, d) = \perp$.

A computation of C is a finite or an infinite sequence $\lambda = s_0, s_1, s_2, ...$ of states such that for $i \geq 0$, s_{i+1} is a successor of s_i if s_i is not a dead state. The length of λ, $|\lambda|$, is ω if λ is infinite, and the number of states minus 1 if λ is finite. To have a uniform notation for both finite and infinite computations, 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 $\leq -\{(\omega, \omega)\}$. Let Γ denote the set of all computations. For a $\lambda \in \Gamma$, we refer to a sub-computation starting at state s over λ as an s-computation, denoted by $\lambda(s)$. Note that $\lambda(s) \in \Gamma$ is also a computation. For any computation $\lambda \in \Gamma$ and indexes $0 \leq i \preceq j \preceq |\lambda|$, we use $\lambda[i]$, $\lambda[0, i]$, $\lambda[i, |\lambda|]$, and $\lambda[i, j]$ to denote the i-th

state in λ, the finite prefix s_0, s_1,...,s_i of λ, the suffix s_i, s_{i+1}, ... of λ, and an interval s_i,..., s_j of λ respectively.

We now define the notion of strategies over concurrent game structures. A strategy for an agent $a \in \mathcal{A}$ is a function f_a that maps a nonempty finite state sequence $\lambda \in S^+$ to a state in S by $f_a(\lambda) = \tau(s, d_a)$ if the last state s in λ is not a dead state. Thus the strategy f_a for agent a induces a set of computations that a can enforce from the initial state s_0 in a CGS C. We define the outcomes of f_a from state s to be the set $out(s, f_a)$ of s-computations that agent a enforces when it follows the strategy f_a. Similarly, Given a set $A \subseteq \mathcal{A}$ of agents, a strategy for the agents in A is a function f_A that maps a nonempty finite state sequence $\lambda \in S^+$ to a state in S by $f_A(\lambda) = \tau(s, d_A)$ if the last state s in λ is not a dead state. Thus the strategy f_A for the set of agents A induces a set of computations that the agents in A can enforce from the initial state s_0 in a CGS C. We define the outcomes of f_A from state s to be the set $out(s, f_A)$ of s-computations that the set of agents A enforce collaboratively when they follow the strategy f_A.

Example 1. We now show how the train packing problem can be modeled by a concurrent game structure.

The train packing problem can be described as follows and depicted as shown in Fig. 1 (1). Suppose A_1, A_2 and A_3 are respectively sets of three kinds of different goods g_1, g_2 and g_3. For each piece of goods g_i, $1 \leq i \leq 3$, the weight of g_i is a_i and the number of g_i is n_i. Also we have a train consisting of a sequence of carriages C_1, C_2, C_3, ... with capacity c_1, c_2, c_3, $\cdots$ respectively. Further, three workers P_1, P_2 and P_3 are employed to load the train but P_i is required to only carry goods g_i from A_i into any carriages from head to tail one with the priority of near head first in order to bite off the extra carriages. Thus, the nearer to the head the carriage is, the earlier installed it is.

Let $C_i.A_j$, $1 \leq i \leq n$, $1 \leq j \leq 3$, denote the weight of a piece of goods from A_j in carriage C_i, and $A = \{P_1, P_2, P_3\}$ be the set of workers (agents). For each worker P_i, at each state, P_i can make a decision in $\Delta^{P_i} = \{1, 0\}$ to carry a piece of goods from A_i into the carriage C_j ($C_j.A_i := C_j.A_i + a_i$) or do nothing. Accordingly, the train packing problem with $n = 3$ is modeled by the CGS as shown in Fig. 1 (2), where a tuple, for instance $(1, 1, 0)$, means that P_1 carried one g_1 and P_2 carried one g_2 from A_1 and A_2

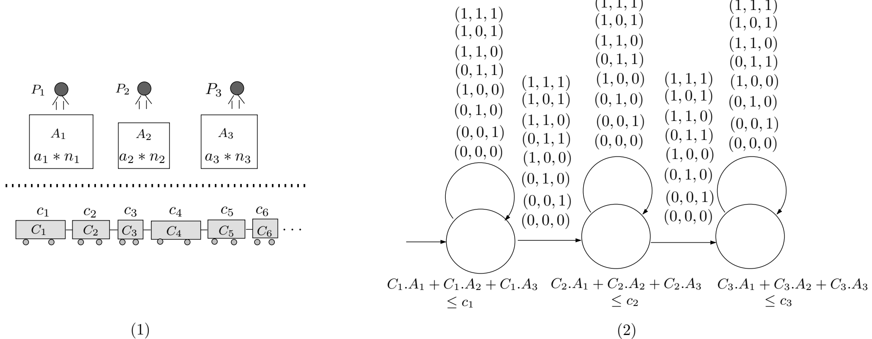


Fig. 1. Train packing problem

respectively into the train concurrently while P_3 did nothing during the time. When the first carriage is full (cannot be loaded further), the goods will be carried into the second carriage, and so on. □

2.2 Automata over Concurrent Game Structures

Automata over Concurrent Game structures (ACGs) are introduced in [18,19] for automata theoretic frameworks of AMC and ATL*. The ACGs are built by extending symmetric automata [2] with sets of agents where the accepting runs are defined by parity and Co-Büchi conditions for AMC and ATL* respectively. In this section, we introduce alternating Büchi automata over Concurrent Game structures (BCGs) by generalizing alternating Büchi automata [20] with sets of agents.

With traditional automata theory, nondeterminism gives a computing device the power of existential choice. Its dual gives a computing device the power of universal choice. Motivated by this, alternating automata where both existential choice and universal choice permitted are proposed in [22,23]. To introduce alternation transitions into automata, positive Boolean formulas built from the set of states are useful. For a given set X of states, $\mathbb{B}^+(X)$ means the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$). We say that $Y \subseteq X$ satisfies a formula $\theta \in \mathbb{B}^+(X)$ if the truth assignment that assigns *true* to the members of Y and $false$ to the members of $X - Y$ satisfies θ. For example, suppose $X = \{q_1, q_2, q_3, q_4\}$, the set $\{q_1, q_2\}$ and $\{q_3, q_4\}$ both satisfy the formula $q_1 \wedge q_2 \vee q_3 \wedge q_4$, while the set $\{q_1, q_4\}$ does not. Accordingly, alternating Büchi automata are formally defined below [20].

Definition 2. An Alternating Büchi Automaton (ABA) is a tuple $A = (\mathcal{P}, Q, q_0, \delta, F)$, where $\mathcal{P}$ is a finite alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \in 2^Q$ is a set of final states, and $\delta : Q \times \mathcal{P} \rightarrow \mathbb{B}^+(Q)$ is a transition function that maps a state and an input letter to a positive boolean combination of states. □

Because of the universal choice in alternating transitions, a run of an alternating automaton is an infinite tree rather than an infinite sequence.

Let N be the set of positive integers and a variable, say x, denote an element of N^*. A tree [24] is a finite or infinite nonempty set $T \subseteq N^*$, such that if $x.i \in T$, where $x \in N^*$ and $i \in N$, then $x \in T$, and further if $i > 1$ then $x.(i-1) \in T$. The elements of T are called nodes. We often omit '.' if it is clear in the context. If x and xi are nodes of T, then x is the parent of xi and xi is the child of x. The node x is a leaf if it has no children. By the definition, the empty sequence ϵ is a member of every tree and called the root of a tree. A branch (really path) of T is a subset $\pi \subseteq T$ such that $\epsilon \in \pi$ and for each $x \in \pi$ either x is a leaf or there is a unique i such that $x \cdot i \in \pi$. For instance, as depicted in Fig.2 (1), $T = \{\epsilon, 1, 2, 3, 1.1, 1.2, 1.3, 3.1, 3.2, 3.3\}$ is a tree and $\pi = \{\epsilon, 3, 3.2\}$ is a branch of T.

Further, for a finite alphabet Q, a Q-labeled tree is a pair $< T, r >$ where T is a tree and $r : T \rightarrow Q$ maps each node of T to an element in Q. For convenience, for a node $x \in T$, $r(x)$ is called Q label of x. Similarly, for two finite alphabets Q and S, a (Q, S)-labeled tree is a pair $< T, r >$ where T is a tree and $r : T \rightarrow Q \times S$ maps each node of T to an element $(q, s) \in Q \times S$. In this case, for convenience, we use $r(x)[1]$ and $r(x)[2]$ to

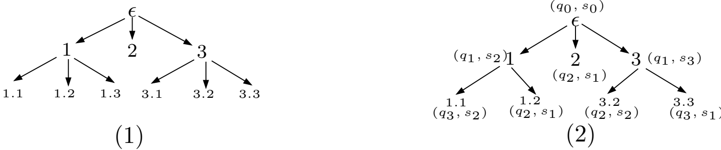


Fig. 2. Tree and (Q, S)-labeled tree

denote the Q and S label of node $x \in T$ respectively. A (Q, S)-labeled tree is illustrated in Fig.2 (2).

A run of an ABA A on an infinite word $w = w_0, w_1, w_2, ...$ is a Q-labeled tree $< T, r >$ such that $r(\epsilon) = q_0$, and for each i, if $|x| = i$, $r(x) = q$, and $\delta(q, w_i) = \theta$, then x has k children x_1, ..., x_k, for some $k \leq |Q|$, and $\{r(x_1), ..., r(x_k)\}$ satisfies θ. For example, if $\delta(q_0, w_0) = q_1 \wedge q_2 \vee q_3 \wedge q_4$, then the possible runs of A on w have a root labeled q_0, and two nodes in level one labeled q_1 and q_2 or q_3 and q_4 respectively.

A run tree is accepted iff all branches (paths) in the run tree satisfy the Büchi condition that there exist infinitely many positions on the branch labeled with the final state. A word w is accepted iff there exists an accepting run on it.

ABAs are built based on Kripke structures or Transitions systems. To well characterize open systems, we further generalize ABAs to concurrent game structures, named alternating Büchi automata over Concurrent Game structures (BCGs).

Definition 3. An alternating Büchi automaton over Concurrent Game structure (BCG) is a tuple $B = (\mathcal{P}, \mathcal{A}, Q, q_0, \delta, F)$, where $\mathcal{P}$ is a finite alphabet; $\mathcal{A}$ is a finite set of agents; Q is a finite set of states; $q_0 \in Q$ is the initial state; $F \in 2^Q$ is a set of final states; and $\delta : Q \times \mathcal{P} \times 2^{\mathcal{A}} \rightarrow \mathbb{B}^+(Q)$ is a transition function that maps a state, an input letter and a set of agents to a positive boolean combination of states. □

Let $\lambda = s_0, s_1, s_2, ...$ with $s_{i+1} = \tau(s_i, d)$ for each i be an infinite computation in CGS $C = (\mathcal{P}, \mathcal{A}, S, s_0, l, \Delta, \tau)$. A run of a BCG B on λ is an (Q, S)-labeled tree $< T, r >$ such that:

- $r(\epsilon) = (q_0, s_0)$, and
- for each i, if $|x| = i$, $r(x) = (q_i, s_i)$, and $\delta(q_i, l(s_i), A) = \theta$, then x has k children x_1, ..., x_k, for some $k \leq |Q|$, $\{r(x_1)[1], ..., r(x_k)[1]\}$ satisfies θ and $r(x_1)[2] = ... = r(x_k)[2] = s_{i+1}$, and there exists $d_A \in \Delta(s_i)$ such that $\tau(s_i, d_A) = s_{i+1}$.

A run tree is accepted iff all branches in the tree satisfy the Büchi condition. A computation λ is accepted if there exists a run tree on λ which is acceptable in B. Accordingly, a BCG can accept a CGS, if all computations in the CGS are acceptable by the BCG; and a BCG is empty if no run trees in the BCG are acceptable.

Generalized Büchi Automata (GBA) are often immediate results of the transformations from temporal logics to automata [21]. We also present its alternating version over concurrent game structures which will be used later.

Definition 4. A Generalized alternating Büchi automaton over Concurrent Game structure (GBCG) is a tuple $GB = (\mathcal{P}, \mathcal{A}, Q, q_0, \delta, F = \{F_1, ..., F_n\})$, where $\mathcal{P}$, $\mathcal{A}$, Q, q_0 and δ are the same as the ones in BCGs; F is a set of accepting sets $\{F_1, ..., F_n\}$, $n \geq 0$, and $F_i \subseteq Q$ for each $0 \leq i \leq n$. □

The run trees of GBCGs are the same as the ones for BCGs. However, the accepting conditions are different. A run tree of a GBCG *GB* is accepted iff for any acceptance set $F_i \in F$, on each path of the tree, at least one state in F_i appears infinitely often. A computation λ is accepted if there exists a run tree on λ which is acceptable in *GB*. Accordingly, a GBCG can accept a CGS, if all computations in the CGS are acceptable; and a GBCG is empty if no run trees in the GBCG are acceptable. An example for GBCG can be found later on.

3 Alternating Interval Based Temporal Logics

Alternating interval based temporal logics extend propositional interval based temporal logics with modal operators that express an agent or a coalition of agents to have a strategy to accomplish a goal.

3.1 Alternating Interval Temporal Logic

Alternating Interval Temporal Logic (AITL) formulas can be defined by the following grammar:

$$P ::= p \mid \neg P \mid P \vee Q \mid \bigcirc_{\langle A \rangle} P \mid P ;_{\langle A \rangle} Q$$

where $p \in \mathcal{P}$ is an atomic proposition and $A \subseteq \mathcal{A}$ is a subset of agents. $\bigcirc_{\langle A \rangle}$ and $;_{\langle A \rangle}$ are basic temporal operators with path quantifier $\langle\rangle$.

AITL formulas are interpreted over CGSs $C = (\mathcal{P}, \mathcal{A}, S, s_0, l, \Delta, \tau)$. A computation $\lambda(s)$ starting from a state s in a concurrent game structure C satisfies the AITL formula P, denoted by $\lambda(s) \models P$. The satisfaction relation ($\models$) is inductively defined as follows:

- $\lambda(s) \models p$ for propositions $p \in \mathcal{P}$, iff $p \in l(s)$
- $\lambda(s) \models \neg P$, iff $\lambda(s) \not\models P$
- $\lambda(s) \models P \vee Q$, iff $\lambda(s) \models P$ or $\lambda(s) \models Q$
- $\lambda(s) \models \bigcirc_{\langle A \rangle} P$, iff $|\lambda(s)| \geq 2$, and there exists a strategy f_A for the agents in A, such that $\lambda(s) \in out(s, f_A)$, and $\lambda(s)[1, |\lambda|] \models P$
- $\lambda(s) \models P ;_{\langle A \rangle} Q$, iff there exists a strategy f_A for the agents in A, we have $\lambda(s) \in out(s, f_A)$, and there exists $0 \leq i \leq |\lambda(s)|$, such that $\lambda(s)[0, i] \models P$ and $\lambda(s)[i, |\lambda(s)|] \models Q$

Note that AITL is an extension of PITL with path quantifiers. As special cases, $\langle \Sigma \rangle$ and $\langle \emptyset \rangle$ corresponds to existential $\exists$ and universal $\forall$ path quantification, respectively. When $A = \emptyset$, the formulas are in fact PITL formulas. As the dual of $\langle\rangle$, we use $[]$, defined by $[] \stackrel{\text{def}}{=} \langle \Sigma \setminus A \rangle$.

The abbreviations *true*, *false*, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. Also, some useful derived constructs such as $\odot_{\langle A \rangle}$ (weak next), $\Box_{\langle A \rangle}$ (always), $\Diamond_{\langle A \rangle}$ (sometimes), *empty* (a computation with zero length), *more* (the current state is not the final one over a finite computation), *halt*(p) (holds over a finite computation if and only if p is true at the final state), *fin*(p) (holds as long as p is true at the final state), and *keep*(p) (holds over a finite computation as long as p holds at all states ignoring the last one) can also be defined.

3.2 Alternating Projection Temporal Logic

To describe some interesting properties, we also extend Propositional Projection Temporal Logic (PPTL) to Alternating Projection Temporal Logic (APTL) which can be obtained by including a projection construct to AITL.

$$P ::= p \mid \neg P \mid P \vee Q \mid \bigcirc_{\langle A\rangle} P \mid (P_1, ..., P_m) prj_{\langle A\rangle}\ Q$$

To define the semantics of the projection operator we need an auxiliary operator for computations. Let $\lambda = s_0, s_1, ...$ be a computation, and r_1, ..., r_h be integers ($h \geq 1$) such that $0 = r_1 \leq ... \leq r_h \leq |\lambda|$. The projection of λ onto r_1, ..., r_h is the computation, $\lambda \downarrow (r_1, ..., r_h) = s_{t_1}, s_{t_2}, ..., s_{t_l}$ where t_1, ..., t_l are obtained from r_1, ..., r_h by deleting all duplicates. That is, t_1, ..., t_l is the longest strictly increasing subsequence of r_1, ..., r_h. For example, $s_0, s_1, s_2, s_3, s_4 \downarrow (0, 0, 2, 2, 2, 3) = s_0, s_2, s_3$.

Accordingly, the semantics of $prj_{\langle A\rangle}$ operator is defined, as before, relative to a computation starting from s, $\lambda(s)$:

- $\lambda(s) \models (P_1, ..., P_m) prj_{\langle A\rangle}\ Q$, iff there exists a strategy f_A for the agents in A, and $\lambda(s) \in out(s, f_A)$, and integers $0 = r_0 \leq r_1 \leq ... \leq r_m \leq |\lambda(s)|$ such that $\lambda(s)[r_{i-1}, r_i] \models P_i$, $0 < i \leq m$ and $\lambda \models Q$ for one of the following λ:
  (a) $r_m < |\lambda(s)|$ and $\lambda = \lambda(s) \downarrow (r_0, ..., r_m) \cdot \lambda(s)[r_m + 1, ..., j]$ or
  (b) $r_m = j$ and $\lambda = \lambda(s) \downarrow (r_0, ..., r_m)$ for sone $0 \leq h \leq m$

Note that in APTL, $;_{\langle A\rangle}$ can be derived from $prj_{\langle A\rangle}$ by $P;_{\langle A\rangle} Q \stackrel{\text{def}}{=} (P, Q) prj_{\langle A\rangle}\ empty$. So, APTL is an extension of AITL. Fig.3 shows the possible semantics of $(P_1, P_2)\ prj_{\langle A\rangle}\ Q$. Here Q and P_1 start to be interpreted at state s_0; subsequently, P_1 and P_2 are interpreted sequentially; Q is interpreted in parallel with $P_1;_{\langle A\rangle} P_2$ over the interval consisting of endpoints of subintervals over which P_1 and P_2 are interpreted. Three possible cases are as follows: (a) P_2 terminates before Q; (b) Q and P_2 terminate at the same state; (c) Q terminates before P_2.

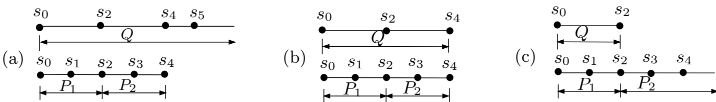


Fig. 3. Semantics of $(P_1, P_2)\ prj_{\langle A\rangle}\ Q$

In order to avoid an excessive number of parentheses, the following precedence rules for AITL and APTL are used where 1 = highest and 5 = lowest.

1	$\neg$
2	$\bigcirc_{\langle A\rangle}$, $\bigcirc_{[A]}$, $\odot_{\langle A\rangle}$, $\odot_{[A]}$, $\Diamond_{\langle A\rangle}$, $\Diamond_{[A]}$, $\Box_{\langle A\rangle}$, $\Box_{[A]}$
3	$\wedge$, $\vee$
4	$\rightarrow$, $\leftrightarrow$
5	$;_{\langle A\rangle}$, $;_{[A]}$, $prj_{\langle A\rangle}$, $prj_{[A]}$

3.3 Specification with AITL and APTL

Now we show how properties of the train packing problem can be specified by AITL and APTL formulas. Let A be the set $\{P_1, P_2, P_3\}$ of the workers (agents or players). For clarity, some useful propositions are defined as follows. $NO_i = C_i.A_1+C_i.A_2+C_i.A_3 \leq c_i$ denotes carriage C_i, $1 \leq i \leq n$, is not overloaded; $F_i = \bigwedge_{1\leq k\leq 3}(c_i - (C_i.A_1 + C_i.A_2 + C_i.A_3) < a_k)$ indicates C_i, $1 \leq i \leq n$, cannot be further loaded; $L_i = C_1.A_i + ... + C_n.A_i = n_i \times a_i$ for $1 \leq i \leq 3$, means all goods in A_i have been loaded into the train while $L = \bigwedge_{1\leq k\leq 3} L_k$ tells us that all goods have been carried into the train. Accordingly, the following properties can be specified by AITL formulas:

1. All goods in A_1, A_2 and A_3 have been sequentially carried into the first n carriages with each carriage being not overloaded.
$$(\Box_{\langle A\rangle} NO_1 ;_{\langle A\rangle} ... ;_{\langle A\rangle} \Box_{\langle A\rangle} NO_n) \wedge fin(L)$$
2. All goods in A_1, A_2 and A_3 have been sequentially carried into the first n carriages with each carriage being not overloaded and no goods can be further carried into.
$$(\Box_{\langle A\rangle} NO_1, ..., \Box_{\langle A\rangle} NO_n) prj_{\langle A\rangle} (len(1) \wedge fin(F_1) ;_{\langle A\rangle} len(1) \wedge fin(F_2) ;_{\langle A\rangle} ... ;_{\langle A\rangle} len(1) \wedge fin(F_{n-1}) ;_{\langle A\rangle} len(1) \wedge fin(L))$$
3. Without considering other goods, all goods in A_1 are eventually carried into the n carriages and each carriage is not overloaded.
$$(\Box_{\langle P_1\rangle} NO_1 ;_{\langle P_1\rangle} ... ;_{\langle P_1\rangle} \Box_{\langle P_1\rangle} NO_n) \wedge fin(L_1)$$
4. Without considering other goods, all goods in A_1 are eventually carried onto the n carriages with each carriage being not overloaded and no goods can be carried into.
$$(\Box_{\langle P_1\rangle} NO_1, ..., \Box_{\langle P_1\rangle} NO_n) prj_{\langle P_1\rangle} (len(1) \wedge fin(F_1) ;_{\langle P_1\rangle} len(1) \wedge fin(F_2) ;_{\langle P_1\rangle} ... ;_{\langle P_1\rangle} len(1) \wedge fin(F_{n-1}) ;_{\langle P_1\rangle} len(1) \wedge fin(L_1))$$

4 Transformation from APTL to GBCGs

In this section, a transformation from APTL formulas to GBCGs is presented. For simplicity, only the formulas with basic temporal operators, named standard formulas, are considered. Other formulas with derived temporal operators can be equivalently transformed to the standard formulas in advance.

4.1 Normal Forms

For the efficient transformation from APTL formulas to GBCGs, normal forms are defined below.

Definition 5. (Normal Form) Let Q_p be the set of atomic propositions appearing in the APTL formula Q. Normal form of Q can be defined by, $Q \equiv \bigvee_{i=0}^{m} (Q_{ei} \wedge empty) \vee \bigvee_{j=0}^{n} (Q_{cj} \wedge \bigcirc_{\langle A\rangle} Q_j)$ where $Q_{ei}, Q_{cj} \equiv \bigwedge_{k=1}^{l} \dot{q}_k$, $q_k \in Q_p$, $\dot{q}_k$ denotes q_k or $\neg q_k$, and $Q_{ci} \not\equiv Q_{cj}$ if $i \neq j$; each Q_j is an arbitrary APTL formula. □

Definition 6. (Complete Normal Form, CNF) Let Q_p be the set of atomic propositions appearing in an APTL formula Q. The complete normal form of Q is defined by,

$Q \equiv \bigvee_{i=0}^{m}(Q_{ei} \wedge empty) \vee \bigvee_{j=0}^{n}(Q_{cj} \wedge \bigcirc_{\langle A \rangle} Q_j)$ where $Q_{ci}, Q_{cj} \equiv \bigwedge_{k=1}^{l} \dot{q}_k$, $q_k \in Q_p$, $\dot{q}_k$ denotes q_k or $\neg q_k$, and $\bigvee_{j=0}^{n} Q_{cj} \equiv true$ and $\bigvee_{i \neq j}(Q_{ci} \wedge Q_{cj}) \equiv false$; each Q_j is an arbitrary APTL formula. □

Note that a complete normal form is also a normal form, but a normal form may not be a complete normal form since the conditions for complete normal form are stronger than the ones for normal form. Complete normal forms are useful in transforming negation constructs into normal forms.

Moreover, any APTL formula in normal form can be further transformed to its complete normal form. This is formalized and proved in Lemma 1.

Lemma 1. If a formula Q has been transformed to its normal form, then Q can be further transformed to its complete normal form.

The proof also illustrates an constructive algorithm for transforming a normal form into its complete normal form. The following is an example for transforming a normal form into its complete normal form.

Example 2. Transform normal form $p \wedge \bigcirc_{\langle A_1 \rangle} P' \vee q \wedge \bigcirc_{\langle A_2 \rangle} Q'$ to its CNF.

$$p \wedge \bigcirc_{\langle A_1 \rangle} P' \vee q \wedge \bigcirc_{\langle A_2 \rangle} Q' \equiv (p \wedge q) \wedge (\bigcirc_{\langle A_1 \rangle} P' \vee \bigcirc_{\langle A_2 \rangle} Q') \vee (p \wedge \neg q) \wedge \bigcirc_{\langle A_1 \rangle} P' \vee (\neg p \wedge q) \wedge \bigcirc_{\langle A_2 \rangle} Q' \vee (\neg p \wedge \neg q) \wedge \bigcirc_{\langle \emptyset \rangle} false$$ □

Further, if Q is transformed into complete normal form, then $\neg Q$ can be transformed into its normal form. This is proved in Lemma 2.

Lemma 2. If Q is transformed into complete normal form, $Q \equiv \bigvee_{i=0}^{m}(Q_{ei} \wedge empty) \vee \bigvee_{j=0}^{2^n-1}(Q_{cj} \wedge \bigcirc_{\langle A \rangle} Q_j)$ then $\neg Q$ can be transformed into its normal form. □

Lemma 3. Let $R \equiv (P_1, ..., P_m) prj_{\langle A \rangle} Q$. Suppose P_1, ..., P_m and Q have been transformed to their normal forms, then R can be rewritten to its normal form. □

Further, Theorem 4 shows that any APTL formula can be transformed into it normal form.

Theorem 4. Any APTL formula can be transformed into its normal form. □

Basically, Lemma 2, Lemma 3, and Theorem 4 are concerned with how a formula can be transformed into its normal form.

4.2 From APTL to Normal Form Graphs

Roughly speaking, Normal Form Graph (NFG) of an APTL formula is constructed by repeatedly decomposing the APTL formula and the new generated formulas to the current and next states according to normal forms. The general idea for constructing NFGs is as follows. To construct NFG of Q, initially, a root node Q is created. Then we transform Q to its normal form. Suppose $Q \equiv r \wedge empty \vee p \wedge r \wedge \bigcirc_{\langle A_1 \rangle}(P' \vee Q') \vee q \wedge$

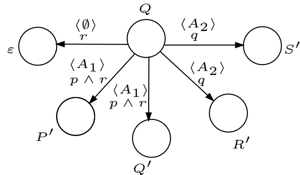


Fig. 4. Constructing NFGs

$\bigcirc_{\langle A_2\rangle}(R' \wedge S')$. As illustrated in Fig.4, five new nodes: ε, P', Q', R' and S' are created; and also the following relations between the new created nodes and the old ones are produced: $\delta(Q, r, \langle\emptyset\rangle) = \varepsilon$, $\delta(Q, p \wedge r, \langle A_1\rangle) = P' \vee Q'$, $\delta(Q, q, , \langle A_2\rangle) = R' \wedge S'$. Note that ε is a node without successors (named dead node).

Thus, to construct the whole NFG, P', Q', R' and S' need to be treated in a similar way. Note that when creating a node P, the new node will be added if node P does not already exist, otherwise only an edge back to the existing node P is added. The formal definition of NFG is given below. Note that, for an APTL formula R, $D(R) = \{R_1, ..., R_n\}$ if $R \equiv R_1 \ddagger ... \ddagger R_n$ with $\ddagger$ being $\vee$ or $\wedge$.

Definition 7. (Normal Form Graph, NFG) For an APTL formula P, NFG of P is a directed graph, $G = (Q, q_0, \mathcal{A}, \Sigma, \delta, e)$, where Q denotes a non-empty finite set of nodes with $q_0 \in Q$ being the initial node; $\mathcal{A}$ is the set of all agents; $\Sigma = \{\bigwedge_k \dot{q}_k \mid q_k \in Q_p\}$; $\delta : (Q, \Sigma, 2^{\mathcal{A}}) \rightarrow \mathbb{B}^+(Q)$ is a transition function; $e \in Q$ is an ε node without successors. Further, the set of nodes Q and the transition function δ over the nodes are inductively defined as follows:

1. $q_0 = \{P\}$ and $Q = \{P\}$
2. For each $R \in Q \setminus \{e\}$, if R has not been decomposed, transform R into its normal form: $R \equiv \bigvee\limits_{i=0}^{m}(R_{ei} \wedge empty) \vee \bigvee\limits_{j=0}^{n}(R_{cj} \wedge \bigcirc_{\langle A_j\rangle} R_j)$. Then $Q = Q \cup \{\varepsilon\} \cup \bigcup_{j=0}^{n} D(R_j)$; $e = \varepsilon$; also we have transition relations $\delta(R, \bigvee_{i=0}^{m} R_{ei}, \emptyset) = \varepsilon$ and $\delta(R, R_{cj}, A_j) = R_j$ for each j. □

Let $\lambda = s_0, s_1, s_2, ...$ with $s_{i+1} = \tau(s_i, d)$ for each i be an infinite computation in CGS $C = (\mathcal{P}, \mathcal{A}, S, s_0, l, \Delta, \tau)$. A run of a NFG G on λ is an (Q, S)-labeled tree $< T, r >$ such that:

- $r(\epsilon) = (q_0, s_0)$, and
- for each i, if $|x| = i$, $r(x) = (q_i, s_i)$, and $\delta(q_i, l(s_i), A) = \theta$, then x has k children x_1, ..., x_k, for some $k \leq |Q|$, $\{r(x_1)[1], ..., r(x_k)[1]\}$ satisfies θ and $r(x_1)[2] = ... = r(x_k)[2] = s_{i+1}$, and there exists $d_A \in \Delta(s_i)$ such that $\tau(s_i, d_A) = s_{i+1}$.

Theorem 5 convinces us that for any APTL formula Q, the number of nodes in the NFG of P is finite. This guarantees the termination of the procedure for constructing NFGs of APTL formulas.

Theorem 5. For any APTL formula P, the set of nodes in the NFG of P is finite.

Proof: Similar to the proof for the finiteness of NFGs of PPTL formulas in [4], the proof can be done by induction on the structures of APTL formulas. □

Further, based on the definition of NFG, Algorithm NFG in pseudocode for constructing NFGs of APTL formulas is presented. The algorithm uses *mark*[] to indicate whether or not a formula needs to be decomposed. If $mark[R] = 0$ (unmarked), then R needs further to be decomposed, otherwise $mark[R] = 1$ (marked), thus R has been decomposed or needs not to be done.

Algorithm NFG: Constructing NFGs for APTL formulas

Function NFG(P)
/* precondition: P is an APTL formula*/
/* postcondition: NFG(P) computes NFG of P, $G = (Q, q_0, \mathcal{A}, \delta, e)$*/
begin function
    $Q = \{P\}$; $q_0 = P$; $mark[P]$ ==0;
    while there exists $R \in S$, and $mark[R]$ ==0
        $mark[R]$ =1; rewrite R to its NF, $R \equiv \bigvee_{i=0}^{m}(R_{ei} \wedge empty) \vee \bigvee_{j=0}^{n}(R_{cj} \wedge \bigcirc_{\langle A_j \rangle} R_j)$;
        $Q = Q \cup \{\varepsilon\} \cup \bigcup_{j=0}^{n} D(R_j)$; $e = \varepsilon$;
        $mark[\varepsilon]$ ==1; **for** each $X \in D(R_j)$, $mark[X]$ ==0;
        for each i, $\delta(R, R_{ei}, \emptyset) = \varepsilon$; **for** each j, $\delta(R, R_{cj}, A_j) = R_j$;
    end while
    return G;
End function

Example 3. Constructing the NFG of formula $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$.

By Algorithm NFG, NFG $G = (Q, q_0, \mathcal{A}, \delta, e)$ of formula $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$ can be constructed as shown in Fig.5(1). □

4.3 From NFGs to GBCGs

By the construction of NFGs, a node will be a literal in the disjunction normal form of APTL formulas. According to the semantics of chop construct $P ;_{\langle A \rangle} Q$, to make $P ;_{\langle A \rangle} Q$ satisfiable, P needs to be satisfied by a finite prefix of a computation and Q needs to be satisfied over the suffix. In this case, we call $P ;_{\langle A \rangle} Q$ is fulfilled. Thus, a run resulting from the NFG of an APTL formula P precisely characterizes a model of P iff any chop formulas occurring in the run are fulfilled.

To explicitly indicate whether or not a node (with chop formula) is fulfilled, extra propositions $\mathsf{l_s}$, $s \in N_0$ and $s > 0$, are needed. Let $\mathcal{P}_l = \{\mathsf{l_1}, \mathsf{l_2}, ...\}$ be the set of extra propositions with $\mathcal{P} \cap \mathcal{P}_l = \emptyset$. Note that these extra propositions are merely employed to mark nodes and are not allowed to appear in an APTL formula. When constructing NFGs by using current and future normal forms, for any chop formula $P ;_{\langle A \rangle} Q$, we equivalently write it as $P \wedge fin(\mathsf{l_s}) ;_{\langle A \rangle} Q$. Formally, we have,

$$
\begin{aligned}
& P \wedge fin(l_s);_{\langle A\rangle} Q \\
\equiv\ & (\bigvee_{i=0}^{m}(P_{ei} \wedge empty) \vee \bigvee_{j=0}^{n}(P_{cj} \wedge \bigcirc_{\langle A\rangle} P_j)) \wedge (l_s \wedge empty \vee \bigcirc_{\emptyset} fin(l_s));_{\langle A\rangle} Q \\
\equiv\ & (\bigvee_{i=0}^{m}(P_{ei} \wedge empty) \vee \bigvee_{j=0}^{n}(P_{cj} \wedge \bigcirc_{\langle A\rangle} P_j)) \wedge (l_s \wedge empty \vee \bigcirc_{\emptyset} fin(l_s));_{\langle A\rangle} Q \\
\equiv\ & \bigvee_{i=0}^{m}(P_{ei} \wedge l_s \wedge Q) \vee \bigvee_{j=0}^{n}(P_{cj} \wedge \bigcirc_{\langle A\rangle}(P_j \wedge fin(l_s);_{\langle A\rangle} Q)) \\
\equiv\ & \bigvee_{i=0}^{m}(P_{ei} \wedge l_s \wedge (\bigvee_{i=0}^{m}(Q_{ei} \wedge empty) \vee \bigvee_{j=0}^{n}(Q_{cj} \wedge \bigcirc_{\langle A\rangle} Q_j))) \\
& \vee \bigvee_{j=0}^{n}(P_{cj} \wedge \bigcirc_{\langle A\rangle}(P_j \wedge fin(l_s);_{\langle A\rangle} Q)) \\
\equiv\ & \bigvee_{i=0}^{m}\bigvee_{i=0}^{m}(P_{ei} \wedge Q_{ei} \wedge l_s \wedge empty) \vee \bigvee_{i=0}^{m}\bigvee_{j=0}^{n}(P_{ei} \wedge l_s \wedge Q_{cj} \wedge \bigcirc_{\langle A\rangle} Q_j) \\
& \vee \bigvee_{j=0}^{n}(P_{cj} \wedge \bigcirc_{\langle A\rangle}(P_j \wedge fin(l_s);_{\langle A\rangle} Q))
\end{aligned}
$$

Thus, by using $fin(l_s)$, $P;_{\langle A\rangle} Q$ is fulfilled if there exists an edge with l_s. And $fin(l_s)$ occurring in a node $P \wedge fin(l_s); Q$ means that $P;_{\langle A\rangle} Q$ has not been fulfilled at this node. For convenience, for a node in the form of $P \wedge fin(l_s); Q$, we add an extra label $\tilde{l_s}$ in this node to mean that some chop formula has not been fulfilled at this node. Accordingly, Labeled Normal Form Graph (LNFG) can be defined based on NFG using l_s propositions.

Definition 8. (Labeled Normal Form Graph, LNFG) For an APTL formula P, its LNFG is a tuple $G = (Q, q_0, \mathcal{A}, \delta, e, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$, where Q, q_0, $\mathcal{A}$, δ and e are identical to the ones in NFG, while each $\mathbb{L}_s \subseteq S$, $1 \leq s \leq m$, is the set of nodes with $\tilde{l_s}$ labels. □

Algorithm LNFG: Constructing LNFGs for APTL formulas

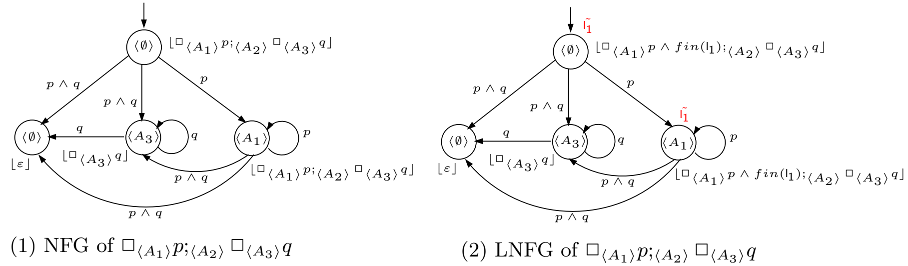
Function LNFG(P)
/* precondition: P is an APTL formula*/
/* postcondition: LNFG(P) computes LNFG of P, $G = (Q, q_0, \mathcal{A}, \delta, e, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$*/
begin function
  $Q = \{P\}$; $q_0 = P$; $mark[P] ==0$; $s = 0$; $\mathbb{L} = \emptyset$; $\mathbb{L} = \mathbb{L} \cup \mathbb{L}_s$;
  while there exists $R \in Q$, and $mark[R] ==0$
    $mark[R] =1$;
    If R is in the form of $P_1;_{\langle A\rangle} P_2$,
      $s = s + 1$; rewrite R as $P_1 \wedge fin(l_s);_{\langle A\rangle} P_2$; $\mathbb{L}_s = \{R\}$;
    rewrite R to its NF,
    $R \equiv \bigvee_{i=0}^{m}(R_{ei} \wedge empty) \vee \bigvee_{j=0}^{n}(R_{cj} \wedge \bigcirc_{\langle A_j\rangle} R_j)$;
    $Q = Q \cup \{\varepsilon\} \cup \bigcup_{j=0}^{n} D(R_j)$; $e = \varepsilon$;
    $mark[\varepsilon] ==1$; **for** each $X \in D(R_j)$, $mark[X] ==0$;
    for each i, $\delta(R, R_{ei}, \emptyset) = \varepsilon$; **for** each j, $\delta(R, R_{cj}, A_j) = R_j$;
    If $X \in D(R_j)$ is in the form of $P_1 \wedge fin(l_s);_{\langle A\rangle} P_2$,
      $\mathbb{L}_s = \mathbb{L}_s \cup \{X\}$;
  end while
  return G;
End function

Algorithm LNFG based on Algorithm NFG is given by further rewriting the chop formulas $P;_{\langle A\rangle} Q$ as $P \wedge fin(l_s);_{\langle A\rangle} Q$ for some $s \in N_0$ whenever a new chop formula is encountered.

A run of an LNFG G on an infinite computation $\lambda = s_0, s_1, s_2, ...$ in a CGS C is the same as the one of an NFG. A run tree is accepted if for each acceptance set $\mathbb{L}_s \in \mathbb{L}$,

there exists at least a state that is not in $\mathbb{L}_s$ and appears in each branch (path) of the run tree infinitely often. A computation λ is accepted if there exists a run tree on λ which is acceptable in G. Accordingly, an LNFG can accept a CGS, if all computations in the CGS are acceptable; and an LNFG is empty if no run trees in the LNFG are acceptable.

Example 4. Constructing LNFG of the APTL formula $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$.



(1) NFG of $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$ (2) LNFG of $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$

Fig. 5. NFG and LNFG of $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$

By Algorithm LNFG, the LNFG of formula $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$ can be constructed as shown in Fig.5 (2). $\Box$

Obviously, an LNFG $G = (Q, q_0, \mathcal{A}, \delta, e, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$ coincides with the accepting condition of the GBCG $GA = (Q, q_0, \mathcal{A}, \delta, F = \{F_1, ..., F_n\})$. Given an LFNG $G = (Q, q_0, \mathcal{A}, \delta, e, \mathbb{L} = \{\mathbb{L}_1, ..., \mathbb{L}_m\})$, an equivalent GBCG $GA = (Q', q'_0, \mathcal{A}', \delta', F = \{F_1, ..., F_n\})$ can be obtained by: $Q' = Q$, $q'_0 = q_0$, $\mathcal{A}' = \mathcal{A}$, $\delta' = \delta \cup \delta(d, \epsilon) = (\langle \emptyset \rangle, d)$ and $F = \{\tilde{\mathbb{L}_1}, ..., \tilde{\mathbb{L}_m}\}$.

The correctness of the transformation is proved in Theorem 6.

Theorem 6. A computation $\lambda = s_0, s_1, s_2, ...$ accepted by the GBCG obtained from APTL formula R precisely characterizes a model of R. $\Box$

Example 5. Constructing GBCG of the APTL formula $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$.

By the transformation from LNFGs to GBCGs, the GBCG of formula $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$ can be constructed as shown in Fig.6 with $F = \{\{\Box_{\langle A_3 \rangle} q, \varepsilon\}\}$. $\Box$

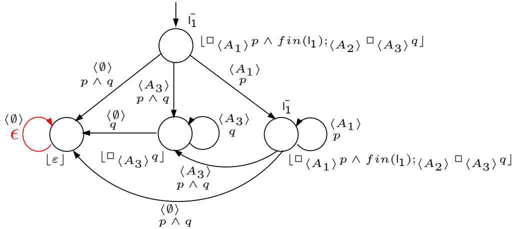


Fig. 6. GBCG of $\Box_{\langle A_1 \rangle} p ;_{\langle A_2 \rangle} \Box_{\langle A_3 \rangle} q$

5 Satisfiability and Model Checking APTL Formulas

Since APTL formulas can be transformed to GBGSs, an approach for checking the satisfiability of an APTL formula Q is achieved by further checking the emptiness of GB of Q. If no runs of a concurrent game structure can be accepted by GB, Q is unsatisfiable, otherwise, Q is satisfiable.

The decision procedure for checking the satisfiability of an APTL formula is presented in Algorithm Check. With this procedure, for a given formula Q, we first construct the GBGS GB of Q, and then we check the emptiness of GB. If no runs can be accepted by GB, Q is unsatisfiable, otherwise, Q is satisfiable.

Further, given an open system modeled by a CGS C, and a desired property specified by an APTL formula P, to check whether or not the system satisfies the desired property is mounting to checking whether or not all computations of C can be accepted by the GBCS of Q. Thus, an automata based model checking approach for APTL is also obtained.

Function Check(P)
/* precondition: P is an APTL formula*/
/* postcondition: Check(P) checks whether or not formula P is satisfiable.*/
begin function
  G =Lnfg(P);
  $GB = Tr(G)$; /* $Tr(G)$ transforms G to a GBGS.*/
  check the emptiness of GB;
  if $L(GB) = \emptyset$, **return** P is unsatisfiable;
  else return P is satisfiable;
end function

6 Conclusion

For the specification and verification of open systems, alternating interval based temporal logics AITL and APTL are presented in this paper. Further, automata based model theory is built for APTL formulas. Based on this, an approach for checking the satisfiability of APTL formulas, and a model checking approach with APTL are proposed.

In the near future, the algorithm for checking the emptiness of a GBCG as well as the algorithm for computing the product of two GBCGs will be investigated. Further, the complexity of the proposed algorithms will be analyzed, and supporting tools will also be developed. In addition, transformation from APTL to GBCGs will be studied for the purpose of verification with APTL.

References

1. Alur, R., Hzenzinger, T.A., Kupferman, O.: Alternating-Time Temporal Logic. Journal of the ACM 49, 672–713 (2002)
2. Wilke, T.: Alternating Tree Automata, Parity Games, and μ-calculus. Bull. Soc. Math. Belg. 8(2) (May 2001)
3. Moszkowski, B.: Reasoning about digital circuits, Ph.D Thesis, Department of Computer Science, Stanford University, TRSTAN-CS-83-970 (1983)

4. Duan, Z., Tian, C., Zhang, L.: A Decision Procedure for Propositional Projection Temporal Logic with Infinite Models. Acta Informatica 45(1), 43–78 (2008)
5. Pnueli, A.: The temporal logic of programs. In: Proc. 18th IEEE Symp. Found. of Comp. Sci., pp. 46–57 (1977)
6. Clark, M., Gremberg, O., Peled, A.: Model Checking. The MIT Press, Cambridge (2000)
7. Vardi, M.Y.: Verification of Open Systems. In: Ramesh, S., Sivakumar, G. (eds.) FST TCS 1997. LNCS, vol. 1346, pp. 250–266. Springer, Heidelberg (1997)
8. de Alfaro, L.: Game Models for Open Systems. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 269–289. Springer, Heidelberg (2004)
9. Ben-Ari, M., Manna, Z., Pnueli, A.: The temporal logic of branching time. Acta Informatica 20, 207–226 (1983)
10. Clarke, E.M., Emerson, E.A.: Desigh and syntesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) LP 1981. LNCS, vol. 131. Springer, Heidelberg (1981)
11. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science 27, 333–354 (1983)
12. Emerson, E.A.: Temporal and Modal Logic. Computer Science Department, University of Texas at Austin, USA (1995)
13. Moszkowski, B.: Reasoning about digital circuits. Ph.D Thesis, Department of Computer Science, Stanford University. TRSTAN-CS-83-970 (1983)
14. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD thesis, University of Newcastle Upon Tyne (May 1996)
15. Duan, Z., Tian, C.: A Unified Model checking Approach with Projection Temporal Logic. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
16. Tian, C., Duan, Z.: Model Checking Propositional Projection Temporal Logic Based on SPIN. In: Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 246–265. Springer, Heidelberg (2007)
17. Tian, C., Duan£, Z.: Propositional projection temporal logic, buchi automata and omega-regular expressions. In: Agrawal, M., Du, D.-Z., Duan, Z., Li, A. (eds.) TAMC 2008. LNCS, vol. 4978, pp. 47–58. Springer, Heidelberg (2008)
18. Schewe, S., Finkbeiner, B.: Satisfiability and Finite Model Property for the Alternating-Time mu-Calculus. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 591–605. Springer, Heidelberg (2006)
19. Schewe, S.: ATL* Satisfiability Is 2EXPTIME-Complete. ICALP (2), 373–385 (2008)
20. Vardi, M.Y.: Alternating Automata and Program Verification. In: van Leeuwen, J. (ed.) Computer Science Today. LNCS, vol. 1000, pp. 471–485. Springer, Heidelberg (1995)
21. Katoen, J.-P.: Concepts, Algorithms, and Tools for Model Checking. Lecture Notes of the Course Mechanised Validation of Parallel Systems (1999)
22. Brzozowski, J.A., Leiss, E.: Finite automata, and sequential networks. Theoretical Computer Science 10, 19–35 (1980)
23. Chandra, A.K., Kozen, D.C., Stockmeyer, L.J.: Alternation. Journal of the Association for Computing Machinery 28(1), 114–133 (1981)
24. Vardi, M.Y.: Nontraditional applications of automata theory. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 575–597. Springer, Heidelberg (1994)

Author Index

Abdelhalim, Islam 371
Arbab, Farhad 106

Back, Ralph-Johan 24
Basu, Samik 338
Bayley, Ian 630
Bejleri, Andi 270
Ben-Hafaiedh, Imene 436
Billington, Jonathan 420

Chane-Yack-Fa, Raphaël 581
Changizi, Behnaz 106
Cheng, Bin 518
Chin, Wei-Ngan 171, 468
Chossart, Romain 581
Craciun, Florin 171

Damljanovic, Danica 237
Ding, Zuohua 138, 155
Duan, Zhenhua 90, 694
Dwyer, Matthew B. 21

Ehlers, Rüdiger 565
Eriksson, Johannes 24
Ezekiel, Jonathan 549

Fehnker, Ansgar 485
Fraikin, Benoît 581
Frappier, Marc 581
Futatsugi, Kokichi 1, 501

Gallasch, Guy Edward 420
Gao, Ping 355
Gerke, Michael 565
Goriac, Eugen-Ioan 220
Graf, Susanne 436

Hanna, Youssef 338
Hatebur, Denis 253
He, Guanhua 171, 468
Heisel, Maritta 253
Hoover, H. James 188
Huuck, Ralf 485

Johnsen, Einar Broch 646

Kaliappan, Prabhu Shankar 613
Kokash, Natallia 106
König, Hartmut 613

Larsen, Peter Gorm 40
Li, Xin 188
Liu, Jing 155
Liu, Shaoying 662
Liu, Yang 388, 518
Lomuscio, Alessio 204, 549
Lucanu, Dorel 220
Luo, Chenguang 171, 468

MacCaull, Wendy 122
Madlener, Ken 287
Mastroeni, Isabella 452
Meseguer, José 303
Meyer, Bertrand 597
Miao, Huaikou 662

Nakamura, Masaki 678
Nanz, Sebastian 597
Nikolić, Đurica 452

Ogata, Kazuhiro 501, 678
Ölveczky, Peter Csaba 303
Ouenzar, Mohammed 581
Owe, Olaf 646

Peter, Hans-Jörg 565
Ping, Jing 138
Poernomo, Iman 56
Pu, Geguang 138

Qin, Shengchao 171, 468
Quinton, Sophie 436

Rabbi, Fazle 122
Rajan, Hridesh 338
Reif, Wolfgang 485
Ribeiro, Augusto 40
Roşu, Grigore 220
Rudnicki, Piotr 188

Sakallah, Karem 404
Samuelson, David 338
Sánchez, Alejandro 74
Sánchez, César 74
Schlatte, Rudolf 646
Schmerl, Sebastian 613
Schneider, Steve 371
Sharp, James 371
Shen, Hui 155
Siirtola, Antti 321
Smetsers, Sjaak 287
Song, Songzheng 388
Strulo, Ben 204
Sun, Jing 237
Sun, Jun 388, 518

Tapia Tarifa, Silvia Lizeth 646
Terrell, Jeffrey 56
Tian, Cong 90, 694
Timm, Nils 534
Treharne, Helen 371

van Eekelen, Marko 287
Velev, Miroslav N. 355
Vogelsang, Andreas 485

Walker, Nigel 204
Wang, Hai H. 237
Wang, Hao 122
Wang, Xi 662
Wang, Zheng 138
Wehrheim, Heike 534
West, Scott 597
Wu, Peng 204

Xiao, Hao 138

Yang, Zijiang 404
Yi, Wang 22

Zhang, Min 678
Zhou, Lei 138
Zhu, Hong 630